

RELAZIONE MODULO SOFTWARE TESTING ISW2

Software Testing and Coverage Analysis

Paci Andrea
0286387

INTRODUZIONE

Il report del modulo di *Software testing* ha come obiettivo quello di presentare il lavoro svolto nel testing di due progetti *open source* di Apache: **BookKeeper** e **Syncopé**. Il report è stato stilato in parallelo con la progettazione dei test ed il loro miglioramento, seguendo quindi l'ordine cronologico del lavoro svolto. L'ambiente di lavoro è stato configurato utilizzando **Github** per l'accesso alle repository, **Travis CI** per il building con **Maven** in remoto e **Sonar Cloud** per l'analisi dei test effettuati e le varie **coverages**. I test precedentemente presenti nelle *repository originali* sono stati eliminati ad eccezione di classi di supporto al testing utilizzate per *inizializzare l'ambiente*: si è infatti scelto di evitare di effettuare **mocking** delle componenti e di replicare l'ambiente di esecuzione *quanto più fedelmente possibile*. Il setup dei progetti è risultato molto complesso, specialmente nel caso di *Syncopé* dove non è stata trovata alcuna documentazione relativa alla configurazione dell'ambiente di lavoro. Per riuscire ad effettuare il setup è stato necessario modificare diversi *pom.xml* e disabilitare *plug-in* non essenziali, in particolare su *Syncopé*.

AMBIENTE DI SVILUPPO

Si è cercato di usare tutti gli strumenti presentati durante il corso: è stato effettuato sia il *building in locale* utilizzando **Maven 3.6** su **MacOSX**, sia *building in remoto* con **Travis CI**. **Travis** è stato configurato per effettuare il building *ad ogni commit*. Per configurare il *building in remoto su Travis* è stato necessario il file *.travis.yml*. Come spiegato nel paragrafo precedente, la fase di setup è stata particolarmente ardua, come conseguenza ci si è ritrovati spesso a modificare i file sorgente e di configurazione di **BookKeeper** e **Syncopé**, arrivando spesso ad una situazione di stallo dove era necessario eliminare il progetto su cui si stava lavorando e riprendere l'originale. Questo ha comportato un utilizzo ridotto del sistema di *Versioning* offerto da **Github** nelle fasi iniziali.

SCelta DELLE CLASSI E DEI METODI

La scelta delle classi e i metodi su cui effettuare testing è stata effettuata seguendo dei criteri.

I criteri per la *scelta delle classi* sono:

- **Metriche dei file:** si è usato il *secondo deliverable* per estendere il calcolo delle metriche fino all'ultima versione rilasciata
- **Stato delle classi:** le classi con un set di attributi limitato e gestibile senza la pregressa conoscenza del sistema.

I criteri per la *scelta dei metodi* sono:

- **Numero di parametri:** i metodi devono esporre almeno un parametro, possibilmente con *un tipo di dato complesso*.
- **Tipi di ritorno ed eccezioni:** i metodi devono avere un tipo di ritorno, lanciare un'eccezione o modificare il valore di almeno uno dei parametri. E' stato posto questo vincolo perché si vuole avere la possibilità di capire l'esito del test in maniera diretta e non attraverso vie alternative (es. modifiche al file system)
- **Branches:** è preferibile che i metodi abbiano *almeno un branch*.
- **Comprensione:** metodi ben documentati, con un comportamento attendibile o predicibile guardando il codice sorgente o il nome delle variabili

Si è cercato quindi di trovare un certo bilanciamento tra *complessità* del metodo da testare e la possibilità di sfruttare le tecniche e i framework affrontati nel corso: sono stati quindi scelti metodo *non banali* come *getter* e *setter*, ma allo stesso tempo con un numero limitato di parametri e branches in modo da poter *testare*

esaustivamente il metodo in esame usando a pieno gli strumenti forniti.

Un'ultima nota prima di procedere nel report: i parametri di input da testare sono stati combinati in maniera **unidimensionale**, questa scelta è avvenuta dopo lo studio dei metodi presi in analisi e si è considerata una buona strada quella di iniziare con una combinazione unidimensionale per poi ampliare i casi di test a seconda dei valori forniti con **coverage** e **mutations**.

Per Bookkeeper sono state scelte le classi [org.apache.bookkeeper.BookKeeperAdmin](#) e [org.apache.bookkeeper.bookie.storage.Idb.WriteCache](#)

BOOKKEEPER

Le classi scelte per Bookkeeper sono:

- BookKeeperAdmin
- WriteCache

BookKeeperAdmin è una classe che fornisce metodi per amministrare il cluster di server *bookies*. E' stata scelta poiché è una classe che fornisce funzionalità primarie per l'infrastruttura. Inoltre, come è possibile vedere in figura in fondo al report, nell'ultima release ha avuto ben 43 revisioni e 622 linee di codice aggiunte, rendendola un buon candidato per il testing. La classe **WriteCache** si occupa invece di gestire la cache di scrittura per un *single directory database server*, quindi fornisce funzionalità di scrittura e lettura sulla cache tampone per la scrittura sul database.

Inoltre entrambe le classi sono risultate defective nelle release precedenti.

CATEGORY PARTITION

Per la classe **BookKeeperAdmin** sono stati scelti i seguenti metodi:

- `public static boolean initNewCluster(ServerConfiguration conf) throws Exception`
- `public static boolean nukeExistingCluster(ServerConfiguration conf, String ledgersRootPath, String instanceId, boolean force) throws Exception`
- `public static boolean format(ServerConfiguration conf, boolean isInteractive, boolean force) throws Exception`

Questi metodi per essere eseguiti correttamente hanno bisogno di un ambiente stabilito: *initNewCluster* e *nukeNewCluster* hanno bisogno di un **Cluster Zookeeper** per mantenere i *metadati* dei realtivi *bookies* e *ledgers*, mentre *format*, oltre che di un **Cluster Zookeeper**, ha anche bisogno di un **Cluster Bookkeeper**. La scelta che è stata fatta è quella di fare testing su di un ambiente più vicino possibile ad uno reale, perciò invece di impiegare uso di **Mock**, l'ambiente è stato ricreato in fase di inizializzazione del test. E' stata creata appositamente una classe **entità** chiamata *BookKeeperAdminClusterEntity* per facilitare il passaggio di parametri nella fase di testing.

Il metodo **initNewCluster** serve ad inizializzare un cluster BookKeeper su cui stanno i vari *bookies* contenenti i dati memorizzati. Il cluster di BookKeeper utilizza Zookeeper specialmente per il *sistema di elezione di leader e follower* al fine mantenere la consistenza dei log. Per inizializzare quindi il cluster di BookKeeper è necessario avere un cluster di Zookeeper in funzione, perciò è stata necessaria l'implementazione del cluster Zookeeper nel testing.

Il metodo prende solo un parametro in ingresso: **ServerConfiguration conf**: essendo un tipo di dato complesso, un primo partizionamento è: { null, new ServerConfiguration() }. *ServerConfiguration* è un tipo di dato abbastanza complesso da gestire: estende la classe *AbstractConfiguration<ServerConfiguration>* ed al suo interno sono presenti altri tipi di dato complessi, è quindi ragionevole usare per le fasi iniziali di testing questa partizione, per poi eventualmente espanderla per migliorare la coverage se necessario. Nella documentazione viene esplicitato che nel momento dell'inizializzazione del cluster BookKeeper deve essere già presente un *Zookeeper Cluster* nel path specificato da **ledgersRootpath**, il quale deve essere condiviso tra il cluster di Zookeeper e quello di BookKeeper, e deve essere impostato per la *ServerConfiguration*. Essendo il *ledgersRoothPath* effettivamente un path del file system, si è ritenuto opportuno provare varie possibili combinazioni per un path. Analizzando il codice sorgente di BookKeeper è stata trovata una *configurazione base*: vengono settate alcune proprietà della *ServerConfiguration* con valori presumibilmente di default. Per voler replicare anche un contesto dove la configurazione server non fosse "spoglia" ma

avesse alcune proprietà sono stati aggiunti alcuni casi di test con la configurazione del server segnata come *base*. Non si è però registrato nessun cambiamento significativo nel testing. Il metodo può lanciare un'eccezione se alcune condizioni sono verificate, e per gestire questo possibile esito è stato utilizzato il costrutto *try catch* al fine di dare una corretta interpretazione dell'uscita improvvisa dal metodo.

Il metodo **nukeExistingCluster** effettua esattamente l'opposto di *initNewCluster*: distrugge il cluster di BookKeeper e tutti i bookies/ledger vincolati da esso. Per poter testare questo metodo è necessario avere un ambiente dove sono presenti sia il cluster Zookeeper sia il cluster Bookkeeper: questi vengono inizializzati nella fase di *setup* e vengono distrutti nella fase di *teardown*. Per evitare di effettuare il nuke di un cluster per errore, viene richiesto l'*instanceId*: identificativo del cluster BookKeeper, che se combacia con l'effettivo identificativo del cluster che si vuole distruggere (trovato con il *ledgersRootPath*) viene eliminato, altrimenti no. Se viene impostato a *true* il parametro di *force*, il check sull'ID viene bypassato e il cluster viene distrutto lo stesso. Per gestire la molteplicità di parametri si sceglie di avere un approccio **unidimensionale**: si preferisce creare prima un insieme di test ristretto e semplice, per poi ampliarlo durante le fasi di testing successive riguardanti lo studio della coverage e delle mutazioni. Come per il metodo precedente il parametro **ServerConfiguration conf** è stato inizialmente assunto come settato a *null* o inizializzato con costruttore di *Default* (costruttore vuoto), per poi essere caratterizzato impostando il *ledgersRootPath* e la *configurazione base* esposta poco sopra. Inoltre si è cercato di aggiungere alcuni casi di test in più per provare alcune combinazioni che si ritenevano significative. I casi di test sviluppati sono:

	conf	basicConfig	ledRootPath	retrInstanceId	InstanceId	force
Caso 1:	{ null,	true,	"/ledgers",	true,	"random"	true }
Caso 2:	{new SerConf,	true,	"/ledgers",	true,	"random"	true }
Caso 3:	{new SerConf,	false,	null,	true,	null	false}
Caso 4:	{new SerConf,	true,	"/ledgers",	false,	"random"	true }
Caso 5:	{new SerConf,	false,	"/ledgers",	false,	null	false}
Caso 6:	{new SerConf,	false,	"/wrongPath",	false,	null	true }

Nei casi di test compare la variabile **retrInstanceId**: questo valore booleano indica se l'*instanceId* va recuperato o meno, infatti non può essere fornito a priori, ma deve essere ottenuto dal cluster che viene creato in fase di inizializzazione. Quando il parametro **retrInstanceId** è impostato a *true* allora viene recuperato l'*instanceId* corretto, altrimenti no. Come per l'*inizializzazione del cluster*, anche qui può essere lanciata un'eccezione, quest'ultima viene gestita nello stesso modo.

Il metodo **format** *elimina* i metadati del cluster BookKeeper su Zookeeper. Prima di eliminarli effettua un controllo se sono presenti dei dati all'interno del cluster, e se sono presenti chiede conferma al client se proseguire con l'eliminazione. Nell'eventualità in cui il comando format viene *forzato* con il parametro **force** non viene chiesta nessuna conferma al client e prosegue con l'eliminazione dei metadati. Proprio come per i due metodi precedenti, è necessario avere un determinato ambiente per effettuare il test di questo metodo. Per format l'ambiente viene ricreato diversamente rispetto ai metodi sopra citati: la classe di test *estende* una classe astratta, già presente nel sorgente, che effettua tutto il setup e il teardown. Come parametri di input prende la configurazione server **ServerConfiguration conf**, il **booleano** che decide se fare il prompt della scelta di formattazione del cluster nel caso in cui siano già presenti dei dati e infine il **booleano** che **forza** la formattazione anche in presenza di dati pregressi. Anche qui si è iniziato creando dei casi di test minimali e combinandoli in maniera unidimensionale: i casi di test minimali sono:

	conf	isInteractive	force
Caso 1:	{ null,	true,	false }
Caso 2:	{new SerConf,	false,	true }

Per la classe **WriteCache** sono stati scelti i seguenti metodi:

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

A differenza dei metodi testati per la classe *BookKeeperAdmin* i metodi esposti da *WriteCache* non hanno bisogno di una particolare *inizializzazione* se non della cache stessa. Sono state create due classi entità, una per metodo, per agevolare la creazione di casi di test

Il metodo **put** scrive sulla cache di scrittura una *entry* con un certo *entryId* destinato ad un determinato **Ledger** con il suo *ledgerId*. Nel caso di avvenuta scrittura ritorna come valore **true** e **false** in caso contrario. I primi due paramtri di input sono dei **long** per i quali scegliamo come partizioni { >= 0, < 0 } poichè non si hanno ulteriori informazioni sui range ammissibili ne dalla documetazione nel dal codice stesso, ma è comunque ragionevole affermare che questa è una buona partizione visto che difficilmente si usano ID negativi. Per *ByteBuf* invece, essendo un tipo di dato complesso, abbiamo la classica partizione { null, new *ByteBuf*() }: non sono stati considerati casi diversi da questi due poiche, per come utilizzato all'interno del progetto di *BookKeeper*, l'oggetto *ByteBuf* viene usato in maniera simile ad una stringa. I casi di test inizialmente selezionati sono:

	ledgerID	entryID	entry
Caso 1:	{ -1,	0,	null }
Caso 2:	{ 1,	-1,	new <i>ByteBuf</i> }

Eseguendo dei test sul metodo si è arrivati alla conclusione che la partizione effettuata sui parametri di tipo long era quanto meno sensata: usare ledgerID o entryID con valori negativi comporta un esito negativo del metodo, e quindi il fallimento della scrittura.

L'analogo metodo **get** legge dalla *WriteCache* il contenuto relativo all' *entryID* situato nel Ledger con ID uguale a *ledgerID*. Se viene trovata una entry con i parametri forniti allora verrà *resituito* un *ByteBuf* con il contenuto della entry. Per effettuare la lettura è stato necessario fare una *inizializzazione* dell'ambiente dove veniva istanziato il *WriteCache*, per poi essere riempito con delle entry. Si può effettuare la stessa partizione di prima: per entrambi i parametri selezioniamo come partizioni *minori di zero* e *maggiori-uguali a zero*, ma in questo contesto la scelta più sensata è quella di dividere i casi di test nella partizione {*ID_valido*, *ID_non_valido*} dove con *ID_valido* intendiamo un Id, sia di entry che di ledger, che sia accessibile, o in altre parole, che esiste nella Cache. I casi di test iniziali sono quindi:

	ledgerID_valido	entryID_valido
Caso 1:	{ true,	true}
Caso 2:	{ false,	true}
Caso 3:	{ true,	false}

ADEGUATEZZA E MIGLIORAMENTO DEI CASI DI TEST

In questo paragrafo vengono mostrati per **BookKeeper** i risultati ottenuti successivamente allo studio della **statement & branch coverage** con i relativi miglioramenti ove necessario.

Alcuni metodi presentano il costrutto **lambda** di Java: il calcolo dell'adeguatezza relativa alla coverage prende in considerazione anche quelli.

Per la classe **BookKeeperAdmin** abbiamo:

- **initNewCluster** con una *statement coverage* del 100% e una *branch coverage* non definita
- **nukeExistingCluster** con una *statement coverage* del 100% e una *branch coverage* non definita per il metodo esterno, mentre una *statement coverage* del 61% e una *branch coverage* del 50% per la funzione lambda all'interno.
- **format** con una *statement coverage* del 100% e una *branch coverage* non definita per il metodo esterno e una *statement coverage* del 80% e una *branch coverage* del 83%

Per la classe **WriteCache** abbiamo

- **put** con una *statement coverage* del 94% e una *branch coverage* del 50%
- **get** con una *statement coverage* del 100% e una *branch coverage* del 100%

In fondo vengono riportati degli screenshot per mostrare la coverage dei metodi.

Si è cercato di aggiungere casi di test per aumentare la coverage per tutte le funzioni coinvolte:

- **put**: non si è riusciti ad aumentare la coverage poiché richiede una condizione particolare tra il segmento di dati che viene memorizzato ed il suo offset
- **get** non è possibile migliorarla ulteriormente
- **initNewCluster** non è possibile migliorarla ulteriormente
- **nukeExistingCluster** ha come istruzione/branch non eseguito uno riguardante un *catch*. Nonostante l'aggiunta di test non si è riusciti nel lanciare l'eccezione desiderata

MUTATION E MIGLIORAMENTO DEI CASI DI TEST

Il **mutation test** sulle classi di BookKeeper è stato fatto provando ad utilizzare vari flag per cambiare il tipo ed il numero di mutazioni, producendo risultati molto interessanti. Nel report si è deciso di includere un **mutation test** utilizzando i parametri di default forniti da **PIT** e limitandosi a 50 mutazioni per classe.

Per quanto riguarda la classe **BookKeeperAdmin** si sono trovate delle difficoltà a fare **mutation testing**: ogni metodo testato di BookKeeperAdmin va ad inizializzare un *cluster* sia di Zookeeper che di BookKeeper, ma nel momento in cui avvengono situazioni di **timeout** di connessione o crash del server, che di per se non comportano un problema ai fini del test, nel momento in cui l'ambiente di esecuzione non viene arrestato correttamente le *run di test* successive non riusciranno ad effettuare l'inizializzazione. L'unico test con cui si è potuto fare **mutation** è **BookKeeperAdminFormatTest**. Come è possibile vedere nella figura allegata, le mutazioni sul metodo **format** sono state gestite con una buona robustezza e con solo tre mutazioni sopravvissute.

Invece la classe **WriteCache** non ha presentato gli stessi problemi di BookKeeperAdmin, rendendo possibile un'analisi della reazione alle mutazioni più accurata e una **mutation coverage** più alta.

Il metodo **put** ha avuto **5 mutazioni** SURVIVED durante la fase di test mentre ne ha uccise 5 come è possibile vedere nello screenshot allegato in fondo al report.

Il metodo **get** gestisce molto meglio le mutazioni rispetto a *put*: 3 vengono uccise e solo una sopravvive.

SYNCOPE

Le classi scelte per **Syncope** sono:

- **DefaultPasswordGenerator**
- **RealmUtils**

DefaultPasswordGenerator è una classe che genera password randomiche sulla base di **policies**, le quali offrono un buon livello di personalizzazione nella generazione. La classe **RealmUtils** invece fornisce le funzionalità per gestire i **realms**: delle chiavi utilizzate per gestire l'accesso a risorse condivise e protette. Per queste classi si è scelto di avere un approccio diametralmente opposto rispetto a quello preso per BookKeeper: le classi scelte non hanno avuto un alto numero di commit o di modifiche nell'ultima release ed espongono funzionalità più marginali.

CATEGORY PARTITION

Per la classe **DefaultPasswordGenerator** è stato scelto il metodo

- `public String generate(final List<PasswordPolicy> policies)`

Il quale ritorna la *password* generata sottoforma di *stringa* seguendo le direttive specificate dalla lista di *policies*. Il tipo di dato complesso **List** si può assumere come se fosse un *array* vero e proprio, quindi procediamo chiamando il metodo *generate* come possibili valori:

`policies = { null, Lista vuota, Lista con valori }`

La lista di *policies* viene popolata in fase di testing seguendo altri esempi di impiego di questa funzione all'interno del progetto. Il tipo di dato **PasswordPolicy** espone numerosi attributi booleani che indicano come generare la password, ad esempio se l'ultimo carattere deve essere un numero o un carattere qualsiasi. Per la fase iniziale del test si è optato per fornire una lista di *policies* con valori consentiti senza iterare per tutte le possibili combinazioni.

Per la classe **RealmUtils** sono stati scelti i seguenti metodi:

- `public static String getGroupOwnerRealm(final String realmPath, final String groupKey)`
- `public static Set<String> getEffective(final Set<String> allowedRealms, final String requestedRealm)`
- `public static Set<String> normalize(final Collection<String> realms)`

E' stata implementata anche una classe nel package **entity** chiamata **RealmEntity**: viene usata per gestire in maniera più ordinata e chiara il passaggio di parametri nel testing dei vari metodi.

Il metodo **getOwnerGroupRealm** restituisce una stringa da usare come chiave di accesso per un *Realm* posseduto da un *gruppo*. L'output del metodo è il seguente:

`<realmPath>@<groupKey>`

dove a posto di `<realmPath>` e `<groupKey>` vanno i valori passati come parametro.

Dopo aver ispezionato il codice del metodo si è visto che non contiene nessun tipo di algoritmo basato sulle stringhe passate in input ma ritorna semplicemente la concatenazione dei due parametri intervallati dal simbolo "@". In visione di questo si è ritenuto opportuno testare esclusivamente il caso con stringa nulla e con stringa piena:

`stringInput = { null, Stringa qualsiasi }`

Sono stati quindi implementati in modo unidimensionale due casi di test: uno con entrambe stringhe nulle e uno con entrambe stringhe valide. Effettuando il test è stato notato che la concatenazione tra *null* e una stringa qualsiasi non ritorna una stringa *null*, bensì torna "*null* + '*stringa*' ". Osservato questo comportamento è stato implementato un ulteriore caso di test per approfondire la questione, arrivando ad un totale di 3:

String String

Caso 1: {null, null}
Caso 2: {null, "groupKey"}
Caso 3: {"realmPath", "groupKey"}

Il metodo non lancia eccezioni di alcun tipo, perciò non è stato implementato nessun meccanismo di gestione dell'eccezione.

Per quanto riguarda il metodo **getEffective** non è stato possibile comprendere il motivo di impiego: ne è stata trovata documentazione online nè si è trovata una interpretazione sicura dall'osservazione degli elementi come la segnatura del metodo o il nome delle variabili. Una possibile interpretazione trovata è quella che il metodo effettua un *merge* delle chiavi dei *realms* e ne fornisce il risultato. Nonostante questa impedenza, è stato comunque possibile testare il metodo. Il metodo prende come input un *Set* di stringhe e una *stringa*. Per quanto riguarda le stringhe si può fare una considerazione analoga a quella fatta per il metodo trattato precedentemente, bensì nel codice del metodo è evidente che viene effettuato un *parse* della stringa (in particolare *StartsWithPredicate*), di conseguenza è stato considerato appropriato passare come input anche una stringa vuota con lo scopo di includere casi limite. Per quanto riguarda invece il *Set* di stringhe, per l'uso che se ne fa, si può assumere come un array, e quindi valgono come partizionamenti {null, array vuoto, array con valori}. I casi di test implementati in questa fase sono:

Set<String> String

Caso 1: {null, "AAAA"}
Caso 2: {[""], null}
Caso 3: {[], ""}

Effettuando alcune esecuzioni del metodo si è stati in grado di capire il funzionamento dell'algoritmo implementato, è stato quindi aggiunto un altro caso di test in cui vengono forniti in input parametri "validi":

Set<String> String

Caso 4: {["AAA", ...], "BBBB"}

In questo caso è stato riscontrato che effettivamente si potesse verificare un'eccezione, è stato quindi usato un costrutto *try catch* per gestire questa eventualità e classificare correttamente l'esito del test.

Il metodo **normalize** prende un insieme di stringhe che rappresentano le chiavi di accesso ai *Realms* e le normalizza secondo lo stesso criterio impiegato per gestire l'accesso alla risorsa: dati due insiemi, A e B, con $\#A < \#B$ e sia A che B rappresentano la stessa chiave di accesso (cioè si può accedere al realm usando sia A che B), allora effettuando *normalize(B)* si ottiene A. Questo metodo è infatti usato da **getEffective**. Trattandosi di un *Set<>* come input, si è applicata la solita partizione {null, [], [A, B, ...]}. Gli input testati sono quindi:

Set<String>

Caso 1: {null}
Caso 2: {[]}
Caso 3: {["AAA", ...] }

Ancora una volta il metodo può lanciare un'eccezione ed è stata gestita sempre con un opportuno *try catch*.

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

ADEGUATEZZA E MIGLIORAMENTO DEI CASI DI TEST

In questo paragrafo vengono mostrati per **Syncope** i risultati ottenuti successivamente allo studio della **statement & branch coverage** con i relativi miglioramenti ove necessario.

Per la classe **DefaultPasswordGenerator**, ed in particolare il metodo *generate* è stata ottenuta una **statement coverage** del 100% e una **branch coverage non definita** (causa assenza di branch) senza l'aggiunta di ulteriori casi di test. Anche se si è ottenuto il massimo risultato' stato ritenuto opportuno estendere i casi di test ad un insieme più ampio andando ad aggiungere diverse *policies* in modo da *stressare* il metodo per vederne il comportamento. In particolare sono stati aggiunti i seguenti casi di test: *Tutte le policies abilitate; Singola policy; Policies discordanti tra loro, Policies con dimensioni di massimo e minimo sbagliate, Due policies con dimensioni diverse*. Come effetto collaterale, questi insieme di test hanno aumentato anche la coverage delle altre classi presenti nel sottomodulo

Per la classe **RealmUtils** di hanno invece:

- **getGroupOwner** con una *statement coverage* del 100% e una *branch coverage* non definita (mancanza di branch)
- **normalize** con una *statement coverage* del 100% e una *branch coverage* del 100% per il metodo esterno e sempre *statement* del 100% per la funzione lambda all'interno
- **getEffective** con una *statement coverage* del 100% e una *branch coverage* del 100%

Avendo ottenuto il massimo possibile della coverage per i metodi presi in analisi, si è cercato di aggiungere alcuni casi di test per ottenere la massima coverage anche per il metodo **normalizingAddTo**, riuscendo ad ottenere una *statement coverage* del 91% e una *branch coverage* del 90%. L'unica istruzione che rimane scoperta non si è riusciti a coprirli con nuovi test case *ad-hoc*.

MUTATION E MIGLIORAMENTO DEI CASI DI TEST

Le **classi** testate su **Syncope** hanno prodotto risultati nettamente migliori rispetto a quelle testate su BookKeeper: la **mutation coverage** è nettamente più alta, e nel caso della classe **RealmUtils** raggiunge addirittura l'82%. Nel caso del metodo **getGroupOwnerRealms** vengono uccise **3 mutazioni** e solo 1 ne sopravvive. Il metodo **normalize** si comporta leggermente peggio, probabilmente a causa della sua complessità algoritmica più elevata rispetto a *getGroupOwnerRealms*, ma comunque **5 mutazioni** vengono uccise e solo **3 sopravvivono**. Il metodo **getEffective** è quello che performa peggio di tutti, e non sembra essere una coincidenza che è anche il più complesso tra i 3 metodi analizzati: vengono uccise **2 mutazioni** e **2 sopravvivono**. Nel complesso questo si può assumere come il risultato migliore tra tutte le classi analizzate.

La classe **DefaultPasswordGenerator** ha una buona **mutation coverage** del 33%, il che indica una certa robustezza alle *mutazioni*. L'unico metodo preso in esame è **generate**, e dal report fornito si può evincere che 1 mutazione **viene uccisa** mentre **3 sopravvivono**. Si è provato a fare un affinamento del metodo di testing, ma la run successiva non ha prodotto risultati migliori da quelli già ottenuti.

CONCLUSIONI

L'attività di testing svolta ha prodotto risultati interessanti e spesso inaspettati. E' stato interessante studiare il codice sorgente di un progetto gestito e sviluppato da un team come *Apache*, anche se la complessità dei progetti da testare è stata spesso motivo di rallentamenti nell'attività di testing.

Link SonarCloud Syncope:

https://sonarcloud.io/component_measures?id=andreapaci_syncope&metric=coverage&view=list

Link SonarCloud BookKeeper:

https://sonarcloud.io/dashboard?id=andreapaci_bookkeeper

FileName	Size	Age	Number	Number	Loc_Touch	Loc_Added
bookkeeper-server/src/main/java/org/apache/bookkeeper/client/BookKeeperAdmin.java	1577	452	43	15	999	622
bookkeeper-server/src/main/java/org/apache/bookkeeper/bookie/storage/ldb/WriteCache.java	249	132	3	1	316	311

Metriche calcolate per il progetto BookKeeper

```

    public static boolean format(ServerConfiguration conf,
        boolean isInteractive, boolean force) throws Exception {
        return runFunctionWithMetadataBookieDriver(conf, driver -> {
            try {
                boolean ledgerRootExists = driver.getRegistrationManager().prepareFormat();

                // If old data was there then confirm with admin.
                boolean doFormat = true;
                if (ledgerRootExists) {
                    if (!isInteractive) {
                        // If non interactive and force is set, then delete old data.
                        doFormat = force;
                    } else {
                        // Confirm with the admin.
                        doFormat = IOUtils
                            .confirmPrompt("Ledger root already exists. "
                                + "Are you sure to format bookkeeper metadata? "
                                + "This may cause data loss.");
                    }
                }

                if (!doFormat) {
                    return false;
                }

                driver.getLedgeManagerFactory().format(
                    conf,
                    driver.getLayoutManager());

                return driver.getRegistrationManager().format();
            } catch (Exception e) {
                throw new UncheckedExecutionException(e.getMessage(), e);
            }
        });
    }

    /**
     * Initializes new cluster by creating required znodes for the cluster. If
     * ledgersrootpath is already existing then it will error out.
     *
     * @param conf
     * @return
     * @throws Exception
     */
    public static boolean initNewCluster(ServerConfiguration conf) throws Exception {
        return runFunctionWithRegistrationManager(conf, rm -> {
            try {
                return rm.initNewCluster();
            } catch (Exception e) {
                throw new UncheckedExecutionException(e.getMessage(), e);
            }
        });
    }
}

```

Statement & Branch coverage delle funzioni format e initNewCluster di BookKeeperAdmin

```

public static boolean nukeExistingCluster(ServerConfiguration conf, String ledgersRootPath, String instanceId,
boolean force) throws Exception {
    String confLedgersRootPath = ZKMetadataDriverBase.resolveZkLedgersRootPath(conf);
    if (!confLedgersRootPath.equals(ledgersRootPath)) {
        LOG.error("Provided ledgerRootPath : {} is not matching with config's ledgerRootPath: {}, "
+ "so exiting nuke operation", ledgersRootPath, confLedgersRootPath);
        return false;
    }

    return runFunctionWithRegistrationManager(conf, rm -> {
        try {
            if (!force) {
                String readInstanceId = rm.getClusterInstanceId();
                if ((instanceId == null) || !instanceId.equals(readInstanceId)) {
                    LOG.error("Provided InstanceId : {} is not matching with cluster InstanceId in ZK: {}",
instanceId, readInstanceId);
                    return false;
                }
            }
            return rm.nukeExistingCluster();
        } catch (Exception e) {
            throw new UncheckedExecutionException(e.getMessage(), e);
        }
    });
}

```

Statement & Branch coverage della funzione *nukeExistingCluster* di BookKeeperAdmin

```

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

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

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

```

Statement & Branch coverage della funzione *get* di WriteCache

```

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

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

    long offset;
    int localOffset;
    int segmentIdx;

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

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

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

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

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

```

Statement & Branch coverage della funzione put di WriteCache

```

62.  @Override
63.  public String generate(final List<PasswordPolicy> policies) throws InvalidPasswordRuleConf {
64.      List<DefaultPasswordRuleConf> defaultRuleConfs = new ArrayList<>();
65.
66.      policies.stream().forEach(policy -> policy.getRules().forEach(impl -> {
67.          try {
68.              ImplementationManager.buildPasswordRule(impl).ifPresent(rule -> {
69.                  if (rule.getConf() instanceof DefaultPasswordRuleConf) {
70.                      defaultRuleConfs.add((DefaultPasswordRuleConf) rule.getConf());
71.                  }
72.              });
73.          } catch (Exception e) {
74.              LOG.error("Invalid {}, ignoring...", impl, e);
75.          }
76.      }));
77.
78.      DefaultPasswordRuleConf ruleConf = merge(defaultRuleConfs);
79.      check(ruleConf);
80.      return generate(ruleConf);
81.  }

```

Statement & Branch coverage della funzione generate di DefaultPasswordGenerator

```

public static String getGroupOwnerRealm(final String realmPath, final String groupKey) {
    return realmPath + '@' + groupKey;
}

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

public static Set<String> normalize(final Collection<String> realms) {
    Set<String> normalized = new HashSet<>();
    if (realms != null) {
        realms.forEach(realm -> normalizingAddTo(normalized, realm));
    }
    return normalized;
}

```

Statement & Branch coverage delle funzioni `getGroupOwnerRealm` e `normalize` di `RealmUtils`

```

public static Set<String> getEffective(final Set<String> allowedRealms, final String requestedRealm) {
    Set<String> allowed = RealmUtils.normalize(allowedRealms);
    Set<String> requested = new HashSet<>();
    requested.add(requestedRealm);

    Set<String> effective = new HashSet<>();
    effective.addAll(requested.stream().filter(new StartsWithPredicate(allowed)).collect(Collectors.toSet()));
    effective.addAll(allowed.stream().filter(new StartsWithPredicate(requested)).collect(Collectors.toSet()));

    // includes dynamic realms
    if (allowedRealms != null) {
        effective.addAll(allowedRealms.stream().filter(new DynRealmsPredicate()).collect(Collectors.toSet()));
    }

    return effective;
}

```

Statement & Branch coverage delle funzioni `getEffective` di `RealmUtils` successivamente ad un incremento di casi di test per raggiungere coverage 100%

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
2	16% <div><div></div></div> 82/503	11% <div><div></div></div> 29/264

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
org.apache.bookkeeper.bookie.storage ldb	1	55% <div><div></div></div> 66/121	26% <div><div></div></div> 21/81
org.apache.bookkeeper.client	1	4% <div><div></div></div> 16/382	4% <div><div></div></div> 8/183

Report generale del mutation testing applicato alle classi `WriteCache` e `BookKeeperAdmin`

```

    public static boolean format(ServerConfiguration conf,
                                boolean isInteractive, boolean force) throws Exception {
        return runFunctionWithMetadataBookieDriver(conf, driver -> {
            try {
                boolean ledgerRootExists = driver.getRegistrationManager().prepareFormat();

                // If old data was there then confirm with admin.
                boolean doFormat = true;
                if (ledgerRootExists) {
                    if (!isInteractive) {
                        // If non interactive and force is set, then delete old data.
                        doFormat = force;
                    } else {
                        // Confirm with the admin.
                        doFormat = IOUtils
                            .confirmPrompt("Ledger root already exists. "
                                + "Are you sure to format bookkeeper metadata? "
                                + "This may cause data loss.");
                    }
                }

                if (!doFormat) {
                    return false;
                }

                driver.getLedgerManagerFactory().format(
                    conf,
                    driver.getLayoutManager());

                return driver.getRegistrationManager().format();
            } catch (Exception e) {
                throw new UncheckedExecutionException(e.getMessage(), e);
            }
        });
    }
}

```

Mutation coverage del metodo BookKeeperAdmin.format

```

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

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

        long offset;
        int localOffset;
        int segmentIdx;

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

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

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

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

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

```

Mutation coverage del metodo WriteCache.put

```

2
3     public ByteBuf get(long ledgerId, long entryId) {
4         LongPair result = index.get(ledgerId, entryId);
5         1 if (result == null) {
6             return null;
7         }
8
9         long offset = result.first;
10        int size = (int) result.second;
11        ByteBuf entry = allocator.buffer(size, size);
12
13        1 int localOffset = (int) (offset & segmentOffsetMask);
14        1 int segmentIdx = (int) (offset >>> segmentOffsetBits);
15        entry.writeBytes(cacheSegments[segmentIdx], localOffset, size);
16        1 return entry;
17    }

```

Mutation coverage del metodo WriteCache.get

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
1	89% <div><div>25/28</div></div>	82% <div><div>9/11</div></div>

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
org.apache.syncope.core.provisioning.api.utils	1	89% <div><div>25/28</div></div>	82% <div><div>9/11</div></div>

Report generated by [PIT](#) 1.5.1

Report generale del mutation testing applicato alla classe RealmUtils

Pit Test Coverage Report

Project Summary



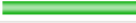





Number of Classes	Line Coverage	Mutation Coverage
1	75% <div><div>118/157</div></div>	33% <div><div>42/126</div></div>

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
org.apache.syncope.core.spring.security	1	75% <div><div>118/157</div></div>	33% <div><div>42/126</div></div>

Report generated by [PIT](#) 1.5.1

Report generale del mutation testing applicato alla classe DefaultPasswordGenerator

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• normalizingAddTo(Set, String)		91%		90%	1	6	1	12	0	1
• getEffective(Set, String)		100%		100%	0	2	0	9	0	1
• normalize(Collection)		100%		100%	0	2	0	4	0	1
• lambda\$normalize\$0(Set, String)		100%		n/a	0	1	0	1	0	1
• getGroupOwnerRealm(String, String)		100%		n/a	0	1	0	1	0	1
Total	4 of 126	96%	1 of 14	92%	1	12	1	26	0	5

Statement & branch coverage della classe **RealmUtils** a seguito di un incremento dei casi di test