## Report per il testing relativo al progetto Apache Zookeeper

### Scelta delle classi

Per scegliere le due classi da testare ho utilizzato i meccanismi studiati con il professor Falessi per la predizione delle classi “buggy” utilizzando il software di Machine Learning Weka. Dal report di Weka sono risultate come potenzialmente buggy (tra le varie classi): Zookeeper.java e DigestAuthenticationProvider.java . Ho deciso di iniziare l’analisi di test sulla classe Zookeeper.java che rappresenta il client attraverso il quale è possibile interagire con uno o più server Zookeeper.

### Category partition per Zookeeper.java

La classe presenta diverse funzionalità da testare, al fine di studiare il SUT ho deciso di selezionare preliminarmente un sottoinsieme di funzionalità da testare, ovvero:

1. Create (permette di istanziare un nodo in Zookeeper);
2. Delete (permette di rimuovere un nodo in Zookeeper);
3. Exists (permette di verificare l’esistenza di un nodo in Zookeeper);
4. getACL (permette di ottenere le ACL per un nodo);
5. getChildren (permette di ottenere i figli di un nodo);
6. getData (permette di ottenere i dati di un nodo);
7. setData (permette di impostare i dati di un nodo);
8. setACL (permette di sovrascrivere le ACL per un nodo).

Per tutte queste funzionalità, la classe esporta due versioni, una sincrona ed una asincrona. Ho deciso di testare esclusivamente le funzionalità sincrone, dal momento che le funzionalità asincrone richiedono testing in ambiente multithread. Per l’analisi del dominio vedere Per l’individuazione dei casi di test a partire da un’analisi del dominio dei parametri coinvolti ho adottato un approccio multidimensionale, producendo con l’aiuto di uno script le matrici con tutte le combinazioni possibili, a partire dai valori che ho individuato come possibili per i parametri coinvolti nei metodi sotto test. In una prima iterazione ho deciso di fare una selezione delle righe presenti nelle matrici andando ad escludere tutte le combinazioni di parametri che possono generare più di un errore, questo perché adottando un approccio black box non ho modo di sapere quale tra le logiche di errore previste venga eseguita prima. Questo infatti dipende dall’ordine in cui lo sviluppatore ha fatto il controllo sui parametri di input ed il successivo throw dell’eccezione. Ad esempio nell’analisi del metodo getData ho escluso gli scenari dove il nodo è inesistente e la versione non è corretta, infatti ciascun valore porta in errore il SUT, generando però una eccezione differente, e non mi è dato modo di sapere, con approccio black box, quale eccezione venga lanciata prima. Dunque ho selezionato i casi di test riportati in Casi di test per la prima iterazione ZookKeeper

### Implementazione dei test per Zookeeper.java

Per implementare i test mi sono affidato ai risultati prodotti dalla fase di category partition, e dunque ho strutturato la mia test suite in modo tale da poter sollecitare il SUT con i casi di test prodotti e verificare che il comportamento del SUT è quello atteso. Dal momento che diverse funzionalità sotto test prevedono una logica di errore diversa (che si riduce al throw di una eccezione con diverso codice di errore) a seconda dell’errore verificatosi, ho deciso di adottare questo meccanismo per l’analisi completa del behaviour:

1. Ho introdotto una enumerazione denominata Type che prevede come valori i possibili scenari di test da eseguire;
2. Ho dunque introdotto uno scenario di test per ogni eccezione differente + lo scenario in cui “tutto va bene” per ogni funzionalità sotto test;
3. Ho poi impostato il metodo configure in modo che tra i vari parametri rientri un valore di tipo Type, per permettere di impostare quale scenario di test si vuole sollecitare con i parametri in input;
4. Ogni metodo di test esegue un Assume.assumeTrue sull’attributo type per verificare che coincide con il “proprio” tipo, ed in quel caso viene eseguito altrimenti viene skippato.

Questo meccanismo mi ha permesso di analizzare ogni possibile scenario. Ho deciso poi di implementare più classi di test, suddividendo in ciascuna classe di test le funzionalità da analizzare, proprio per evitare di rendere troppo complessa la classe a causa di un uso spropositato di questa mia soluzione. Per l’implementazione dei metodi di test si veda:

Per il testing di funzionalità “collegate” tra loro, come create e delete, ho testato nei limiti del possibile le singole funzionalità in maniera atomica, per poi testare anche la loro interazione (ad esempio creando un nodo nel sistema e rimuovendolo successivamente con una delete, per poi verificarne la rimozione con una exists).

Ho deciso di implementare anche dei test di integrazione per verificare la corretta interazione tra Client e Server. Infatti il SUT interagisce con una istanza di server Zookeeper per completare le operazioni richieste. Questa istanza viene creata in un metodo @Before e rimossa in un metodo @After, in modo da avere sempre una istanza pulita nell’esecuzione di ogni test. Dal momento che il corretto funzionamento dei metodi analizzati dipende dalla corretta interazione tra le due classi ho deciso di testarla preliminarmente, utilizzando un mock spia sulla factory del server per verificare che viene effettivamente invocato il metodo registerConnection per registrare la corrente connessione client-server.

N.B. Nell’implementazione dei test d’unità si è assunto corretto il comportamento del server, dunque sono partito dall’assunzione che se la richiesta al server è ben formata, lui la processa, altrimenti restituisce un errore. In questo modo ho potuto testare che il client costruisce correttamente le richieste verso il server.

### Analisi della prima iterazione

Dopo una prima iterazione la coverage del SUT risulta al 26%, che è un buon risultato considerando che abbiamo selezionato solo un sottoinsieme delle funzionalità esportate ed analizzato solo una “versione” di ciascuna funzionalità scelta. Per incrementare la coverage possiamo rivedere il category partition sfruttando il report di JaCoCo per individuare i branch non coperti dai nostri test, e poi incrementare il numero di test spostando la nostra attenzione su altre funzionalità prima escluse.

### Seconda iterazione

In una successiva iterazione ho deciso di adottare un approccio white box, andando ad analizzare i metodi sotto test sfruttando i report di JaCoCo per capire quali sono i branch non coperti e perché. In questo modo ho potuto constatare che i metodi che fanno uso del parametro List acl prevedono una logica di errore nei casi in cui la lista di ACL è vuota o assume il valore null, sollevando una eccezione del tipo “KeeperException…”. Nella prima iterazione non ho notato questo comportamento (non documentato) poiché, sebbene ho stimolato il SUT con valori come null o empty list per questo parametro, non ho osservato comportamenti anomali in quanto il sistema andava già in errore a causa del path mal formato. Dunque ho deciso di riguardare le matrici prodotte nella prima iterazione per selezionare un ulteriore insieme di righe da inserire nel test, in modo da stimolare il sistema correttamente e coprire i branch mancanti. Ho deciso di testare ulteriori funzionalità del SUT in questa seconda iterazione, includendo nei casi di test questi metodi:

1. getEphemerals, metodo utilizzato per ottenere tutti i nodi di tipo ephemeral nel sistema. Per questo metodo esiste una variante (che ho deciso di testare) che prende un parametro in input e permette di ottenere tutti i nodi ephemerals a partire dal path fornito in input;
2. addWatch, metodo utilizzato per aggiungere un Watcher ad un nodo nel sistema;
3. removeWatches, metodo utilizzato per rimuovere tutti i Watcher per uno specifico nodo ed uno specifico tipo (N.B. un Watcher può essere di tipo Any, Data o Children. Data indica che è stato inserito in una operazione di getData, Children che è stato inserito con una operazione di tipo getChildren)
4. removeAllWatches, metodo utilizzato per rimuovere tutti i Watcher di uno specifico tipo dal sistema.

Per tutti questi metodi ho effettuato una ulteriore category partition andando ad individuare tutti i possibili casi di test e comportamenti attesi dal sistema. Per quanto riguarda le funzionalità relative ai Watcher, queste sono state testate implementando un Watcher ad hoc. Di fatti i metodi prevedono una istanza di tipo Watcher, che è un’interfaccia. Le mie implementazioni sono orientate al test per osservare il corretto funzionamento dei metodi, in particolare per le operazioni di addWatch ho previsto un Watcher che memorizza gli eventi ricevuti (ignorando gli eventi null) in un attributo pubblico di tipo List. Dopo aver inserito il watcher sul nodo ho provato a triggerare un evento e verificare che questo veniva correttamente registrato dall’istanza di Watcher. Per le funzionalità di removeWatches ho implementato un Watcher che memorizza solo gli eventi di tipo “SetData” secondo quanto detto prima, e nei metodi di test ho provato a triggerare un evento del tipo “SetData” successivamente alla rimozione del Watcher, per poi verificare che l’evento non fosse stato registrato. Infatti le operazioni di remove si limitano a scollegare il Watcher dall’istanza di nodo e non ad invalidarlo. Per testare le remove con i vari tipi di nodo ho previsto comportamenti differenti per il metodo di test (scelti tramite un if sul parametro che indica il tipo di Watcher) per attaccare il Watcher al nodo in modo che risultasse al sistema di quel tipo. Per i casi di test prodotti dalla category partition vedere … . Ho deciso di migliorare anche il test di integrazione, andando a istanziare una connessione client-server protetta da password ed andando poi a testare (tramite il metodo del client getPassword) che la password impostata corrisponde effettivamente a quella fornita.

Questa seconda iterazione ha migliorato la coverage complessiva portandola ad un 35% e permettendo di raggiungere una mutation coverage del 31% con test strenght dell’87%.

### CategoryPartition per la classe DigestAuthenticationProvider

La classe sotto test presenta pochi metodi pubblici e una scarsa (se non assente) documentazione rispetto alla classe precedentemente testata. Nel testing ho deciso di introdurre in una prima iterazione i seguenti metodi:

1. digest, metodo statico utilizzato per la costruzione di un digest a partire da una stringa in input. Da una ricerca online è emerso che l’algoritmo di digest utilizzato di default è “SHA1”, e da uno sguardo alla documentazione si evince che non esiste un meccanismo esplicitamente riportato per modificare l’algoritmo di digest. Pertanto in questa prima iterazione, seguendo un approccio blackbox, ho deciso di utilizzare l’algoritmo di default; 2)handleAuthentication, metodo utilizzato per inserire informazioni di autenticazione in una connessione client-server. Il metodo dovrebbe costruire il digest a partire da un id fornito (dunque una stringa nel formato user:pass) ed inserire un digest di autenticazione all’interno della classe ServerCnxn fornita in input;
2. isValid, metodo utilizzato per verificare che una stringa sia nel valido formato “id” (ovvero del tipo “user:pass”).

Per questi metodi ho individuato i seguenti possibili parametri:

#### digest

|  |  |
| --- | --- |
| param | values |
| String idPassword | stringa vuota, null, stringa |

#### handleAuthentication

|  |  |
| --- | --- |
| param | values |
| ServerCnxn cnxn | questo parametro non viene fornito in input al test ma ottenuto creando una connessione Client Server al momento. |
| byte[] authData | [], “test:test”.getBytes(), “test”.getBytes() |

#### isValid

|  |  |
| --- | --- |
| param | values |
| String id | stringa vuota, stringa valida, stringa non valida, null |

Ho poi generato questi casi di test utilizzando un approccio unidimensionale:

N.B. per “success” si intende che il metodo sotto test non va in logica di errore.

#### digest

|  |  |
| --- | --- |
| values | expected |
| "" | success |
| null | eccezione null pointer exception |
| “test” | success |

#### handleAuthentication

|  |  |
| --- | --- |
| values | expected |
| [] | eccezione generica (data la scarsa documentazione non ho modo di sapere con approccio black box di che eccezione si tratta) |
| “test:test”.getBytes() | success |
| “test”.getBytes() | eccezione generica (per lo stesso motivo riportato precedentemente) |

###isValid |values|expected| |——|——–| |"“|success e ritorna false| |”test:test“| success e ritorna true| |”test"| success e ritorna false| |null| eccezione di tipo NullPointerException|

### Prima iterazione

In una prima iterazione ho generato metodi di test per coprire i casi di test elaborati nella fase di category partition. Come per la precedente classe ho adottato il meccanismo dell’enumerazione per discriminare quale test eseguire sulla base dei parametri di input. Ho dunque prodotto 5 metodi di test che hanno portato ad una coverage di classe del 72% (vedere …). Dal testing della classe ho compreso che la mia intuizione iniziale sui comportamenti della classe era in realtà errata, in particolare questo è vero per il metodo handleAuthentication che fornisce il risultato “success” in tutti i casi riportati, senza andare in logica di errore. Questo errore è dovuto all’assenza di documentazione e al fatto di aver adottato un approccio prevalentemente black box. Nel testare il metodo digest ho utilizzato come oracolo la classe MessageDigest di java, confrontando il digest prodotto dal SUT con quello prodotto dall’oracolo, utilizzando uno stesso algoritmo di hashing ed uno stesso input, per poi verificare che i due risultati fossero uguali.

Per poter incrementare la coverage decido di adottare in una successiva iterazione un approccio white box, basandomi sul report prodotto da JaCoCo ed analizzando le righe di codice non coperte.

### Seconda iterazione

Adottando un approccio black box ho potuto constatare queste due cose:

1. Il metodo digest utilizza internamente la classe MessageDigest che io ho utilizzato come oracolo, dunque per rendere valido il test è necessario cambiare oracolo;
2. E’ possibile modificare l’algoritmo di hashing previsto accedendo alle property di sistema, in particolare alla property zookeeper.DigestAuthenticationProvider.digestAlg che loro impostano inizialmente al valore “SHA1”, tuttavia nel codice viene utilizzata una variabile privata statica che è inizializzata con tale valore, ed essendo statica viene inizializzata ancor prima che venga istanziata la classe. Di conseguenza non c’è modo di modificarla programmaticamente.

Per il punto 1 ho deciso di non modificare il metodo originale, ma di non aggiungere ulteriori casi di test in cui vado a fornire altri algoritmi di hashing validi. Questo perché, avendo assunto la classe MessageDigest come oracolo, questa risulta essere necessariamente corretta, quindi il suo comportamento sarà sempre corretto indipendentemente dall’algoritmo di hashing fornito. Per quanto detto nel punto due possiamo dichiarare le righe di codice che rientrano nella logica di errore come non raggiungibili, a meno di modificare il comportamento della classe ricorrendo alla reflection. Per incrementare la coverage ho deciso di aggiungere alla mia test suite ulteriori funzionalità della classe sotto analisi che sono state precedentemente ignorate, ovvero:

1. getUsername, metodo utilizzato per ottenere lo username a partire da una stringa id (del tipo “user:pass”);
2. matches, metodo utilizzato per verificare se due stringhe sono uguali.

Un approccio unidimensionale per il category partition ha prodotto questi possibili casi di test

|  |  |  |
| --- | --- | --- |
| method | values | expected |
| matches | “test”,“test” | success e ritorna true |
| matches | “test”,“test2” | success e ritorna false |
| getUsername | “user:pass” | success |
| getUsername | “user” | Eccezione del tipo IndexOutOfBound exception |
| getUsername | "" | Eccezione del tipo IndexOutOfBound exception |

Ho dunque scritto dei metodi per coprire questi casi di test, riuscendo così a raggiungere una coverage del 75% e una mutation coverage del 52% con test strenght del 60% (vedi …)

## Report per il testing relativo al progetto Apache **BookKeeper**

### **Scelta delle classi**

Per scegliere le classi del progetto Apache BookKeeper non ho potuto utilizzare (come in precedenza) l’approccio di predizione basato su Weka, questo perché i dati utilizzati per il training e testing set sono basati sugli Issues riportati su Jira, e gli sviluppatori di BookKeeper hanno deciso di abbandonare Jira come strumento di Issue tracking da diverso tempo, di conseguenza il modello predittivo costruito è inadeguato in quanto non tiene conto degli ultimi anni di sviluppo. Dunque ho scelto le classi arbitrariamente, basandomi su quanto già fatto per Zookkeeper. Ho scelto le classi BookKeeper.java e NetworkTopologyImpl.java.

### **Category partition per Bookkeeper.java**

La classe sotto analisi è il client attraverso il quale è possibile utilizzare le API di BookKeeper per l’interazione con un server. La classe è responsabile della creazione e rimozione di Ledger nel server e della loro gestione (apertura e chiusura). Ho deciso di testare in una prima iterazione queste funzionalità:

1) createLedger, funzionalità che permette di creare un nuovo Ledger nel sistema;

2) deleteLedger, funzionalità che permette di rimuovere un Ledger dal sistema.

Per la prima funzionalità la classe fornisce molteplici implementazioni facendo uso dell’overloading, inoltre fornisce sia una versione sincrona che asincrona. Come fatto per ZookKeeper e per motivi analoghi ho deciso di testare esclusivamente la versione sincrona di queste funzionalità. Per quanto riguarda l’overloading ho scelto di testare il metodo che richiede un maggior numero di parametri formali, questo perché, grazie al grande numero di parametri, permette di individuare molteplici casi di test ed inoltre essendo quello maggiormente configurabile è anche più probabile che contiene possibili bug.

A differenza di ZookKeeper, BookKeeper ha una documentazione meno prolissa e più criptica su alcuni comportamenti, ad esempio non è ben specificato quando e perché può essere lanciata una eccezione, ma viene semplicemente riportato che la createLedger può sollevare una eccezione di tipo BKException. Per quanto riguarda i parametri formali dei metodi, questi sono invece ben descritti.

Per la documentazione relativa a createLedger fare riferimento a: [https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.html#createLedger(int,int,int,org.apache.bookkeeper.client.BookKeeper.DigestType,byte%5B%5D,java.util.Map](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.html" \l "createLedger(int,int,int,org.apache.bookkeeper.client.BookKeeper.DigestType,byte%5B%5D,java.util.Map)).

Per la documentazione relativa a deleteLedger fare riferimento a: [https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.html#deleteLedger(long](https://bookkeeper.apache.org/docs/latest/api/javadoc/org/apache/bookkeeper/client/BookKeeper.html" \l "deleteLedger(long)).

L’individuazione dei domini dei vari parametri è riportata in:

Per la generazione di casi di test ho adottato (per il metodo createLedger) un approccio multidimensionale, costruendo con l’aiuto di uno script una matrice di tutte le combinazioni (riportata in …), a partire dalla quale ho poi selezionato le righe che corrispondono ai casi di test implementati (vedere …).

### Implementazione dei test per BookKeeper.java

A partire dai casi di test individuati nella fase di category partition ho sviluppato vari metodi di test per le funzionalità sotto analisi. Poiché il client deve interagire con un server BookKeeper si è resa necessaria l’istanziazione di un server. Dall’analisi della documentazione è emerso che un server BookKeeper altro non è che un server Zookkeeper su cui sono stati già creati dei nodi corrispondenti ai bookies e ai ledger. Sempre dall’analisi della documentazione ho individuato una classe (org.apache.bookkeeper.util.LocalBookKeeper) che permette di tirare su un server BookKeeper semplice, utilizzabile per il testing. Ho deciso dunque di utilizzare questa classe, assumendola corretta (si noti che il server Zookkeeper è stato già assunto come corretto in precedenza). Dall’esecuzione dei metodi di test ho notato che, diversamente da quanto sperimentato con Zookkeeper, lo start e shutdown del server sono molto costosi in termini di tempo, dunque ho deciso di strutturare la mia test suite in più classi (ad esempio creando una test per l’analisi del throw delle eccezioni e una per l’analisi del metodo in situazioni di successo) piuttosto che utilizzare un approccio basato su Assume. Dovendo eseguire start e shutdown del server per ogni metodo di test (per garantire che ogni test esegua in un ambiente pulito e isolato), l’utilizzo di Assume (che causa lo skip del test, ma che esegue comunque i metodi @Before e @After) risultava troppo oneroso in termini di tempo, in quanto veniva effettuato start e shutdown del server senza che questo venisse effettivamente utilizzato.

Per validare la correttezza del metodo createLedger, ho effettuato la creazione e poi ho utilizzato l’handle ritornato per verificare che i parametri del ledger corrispondessero a quelli forniti in creazione (ad esempio password, metadata e le varie size). Dall’analisi del dominio ho individuato casi di test dove l’ensemble size risulta minore della write quorum size, e per i quali io mi aspettavo un’eccezione (secondo quanto riportato in documentazione), in fase di test ho notato che questa cosa non è vera, anzi il SUT entra in uno stato di wait senza proseguire l’esecuzione. Pertanto ho riportato questi casi di test nel metodo di Parameterized, ma li ho commentati per permettere alla build di passare. Anche per la deleteLedger ho notato che, nel caso in cui il ledger non esiste, non viene sollevata alcuna eccezione. In questo caso il SUT continua normalmente la sua esecuzione. Dal momento che sulla documentazione non viene riportato espressamente il throw di una eccezione in una simile situazione, non posso asserire che si tratti di un bug, se bene non sia a mio parere un approccio corretto.

### **Analisi della prima iterazione**

Al termine della prima iterazione ho effettuato una analisi della coverage con JaCoCo e della mutation coverage con PIT. La line coverage è del 37% mentre la mutation coverage è del 27% con test strength del 82% (vedere …). Per migliorare la coverage ho analizzato il report di JaCoCo, cercando di individuare eventuali branch non coperti dalla mia esecuzione. I soli branch non coperti fanno riferimento ad errori causati da interruzioni del server, che sono difficilmente testabili in quanto richiedono la disconnessione del server durante l’esecuzione del metodo. Pertanto decido di incrementare la coverage in una seconda iterazione aumentando le funzionalità sotto test, andando ad introdurre le funzionalità per aprire un ledger, ovvero: openLedger e openLedgerAdv. Per la category partition vedere …

### **Analisi della seconda iterazione**

Al termine della seconda iterazione ho effettuato una analisi della coverage con JaCoCo e della mutation coverage con PIT. La line coverage raggiunta è del 45% con una mutation coverage del 46% e test strength del 100%. Incrementare il numero di funzionalità sotto test ha effettivamente portato ad un gran miglioramento di queste metriche per la valutazione della bontà della test suite. Decido di terminare qui l’analisi di questa classe, avendo raggiunto una coverage adeguata in riferimento al fatto di aver escluso dal testing le funzionalità asincrone. Nel testing di questa classe ho utilizzato anche ba-dua, ma come per la coverage di JaCoCo non ho avuto modo di incrementare le coppie def-use coperte per ogni singolo metodo variando la selezione dei casi di test.

### **Category partition per NetworkTopologyImpl.java**

La classe NetworkTopologyImpl permette di rappresentare (attraverso una struttura ad albero) una topologia di rete, costituita da più rack e datacenter e nodi che vi sono inseriti. Un nodo viene rappresentato dalla sua network location (un path tra i rack simile ad un path in un filesystem, ad esempio : “/rack1/rack2/rack3”) e il suo identificatore espresso nella forma “host:port”. In una prima iterazione ho deciso di testare le seguenti funzionalità:

1. add, permette di aggiungere un nuovo nodo foglia alla rete (ovvero non un rack);
2. remove, permette di rimuovere un nodo foglia dalla rete;
3. isOnSameRack, permette di verificare, dati due nodi, se appartengono allo stesso rack.

L’analisi del dominio è stata resa semplice dal numero basso di parametri formali richiesti da questi metodi, in particolare i metodi add e remove prevedono esclusivamente un parametro di tipo Node, che è un’interfaccia. Per il parametro Node ho previsto questi possibili valori {null,istanza valida,istanza non valida}. Il metodo isOnSameRack prevede due parametri formali che sono due istanze di Node, per i quali ho previsto i casi in cui: sono entrambi sullo stesso rack, sono su rack diversi, una è null, entrambe sono null. Ho poi generato questi casi di test, utilizzando un approccio multidimensionale per il metodo isOnSameRack:

|  |  |
| --- | --- |
| Method | Values |
| add | Istanza valida |
| add | null |
| add | Istanza non valida |
| remove | Istanza valida |
| remove | null |
| remove | Istanza non valida |
| isOnSameRack | Sono sullo stesso rack |
| isOnSameRack | Non sono sullo stesso rack |
| isOnSameRack | Null, istanza valida |
| isOnSameRack | Null,null |

Istanza valida e istanza non valida sono da intendersi (rispettivamente) come: nodo foglia, nodo non foglia.

### Implementazione dei test per **NetworkTopologyImpl**.java

Ho implementato dei metodi di test per coprire i casi di test individuati nella fase di category partition. In particolare ho analizzato il SUT, per i casi di add e remove, per individuare un corretto funzionamento o il throw di una eccezione (quando attesa). Per testare l’effettivo funzionamento della funzionalità di add ho utilizzato il metodo contains del SUT, che ho supposto come corretto.

Da una prima iterazione su questi test, ho ottenuto una coverage piuttosto bassa in riferimento alla copertura dei singoli metodi sotto test. Ho deciso di adottare un approccio più white box su questi metodi, andando ad analizzare i report di JaCoCo. Così ho potuto individuare alcuni branch non coperti, che corrispondono allo scenario in cui il nodo da aggiungere è di tipo “InnerNode” (una classe definita nel progetto). In tal caso infatti viene effettuato il throw di una eccezione, in quanto non è possibile aggiungere un nodo di questo tipo. Un altro branch non coperto in fase di add corrisponde allo scenario in cui si tenta di inserire nel SUT allo stesso livello di profondità nell’albero un rack ed un nodo foglia. Alla luce di queste informazioni ho rivisto il category partition per i metodi di add e remove, che è stato riportato in …

**Analisi della prima iterazione**

Al termine di questa prima iterazione ho analizzato i report di JaCoCo e pit per la coverage e mutation coverage, per i quali ho ottenuto una line coverage di classe del 39% e una mutation coverage del 23% con test strength del 66%. Per migliorare la coverage della classe ho deciso di incrementare il numero di funzionalità sotto test, andando ad inserire le seguenti:

1. getFirstHalf, permette data una stringa network location di ottenere la prima parte (intesa come la parta a sinistra dell’ultimo delimitatore);
2. getLastHalf, metodo complementare al precedente e permette di ottenere la seconda parte della network location;
3. getLeaves, permette di ottenere tutti i nodi foglia data una network location;
4. getNode, permette di ottenere una istanza di Node a partire da una sua rappresentazione in stringa (network location + identificatore);
5. getNumOfLeaves, permette di ottenere il numero di foglie nel sistema;
6. getNumOfRacks, permette di ottenere il numero di rack nel sistema;
7. getDistance, permette di ottenere la distanza tra due nodi nel sistema.

Per ognuno di questi metodi ho effettuato category partition ed ho individuato i casi di test riportati in …

Per il testing della funzionalità getDistance ho scritto un metodo oracolo che permette di ottenere il valore atteso, basandomi sulla documentazione fornita da BookKeeper, per l’oracolo vedere …

**Analisi della seconda iterazione**

Al termine della seconda iterazione ho ottenuto una line coverage del 43% ed una mutation coverage del 33% con test strength del 70%. Ho analizzato i report di JaCoCo per individuare eventuali branch non coperti, ma sono arrivato alla conclusione che queste righe di codice non possono essere raggiunte dal testing su questa classe. Infatti dall’implementazione si desume che questa classe si aspetta di essere estesa per essere utilizzata in pratica (molti metodi infatti sono metodi “dummy” che devono essere sovrascritti), di conseguenza i developers hanno previsto dei controlli di sicurezza superflui per rendere robusta la classe ad eventuali sotto classi mal scritte. Ad esempio nel metodo getDistance sono effettuati controlli per verificare se i due nodi non appartengono allo stesso livello, tuttavia non c’è modo di inserire due nodi foglia a livelli differenti (vedi analisi del metodo add), dunque non c’è modo di coprire questi branch. Situazioni analoghe si hanno per il metodo add, dove si effettua un controllo per verificare se il nodo rack a cui si vuole assegnare la foglia non è di tipo “InnerNode”, tuttavia non c’è modo di inserire nella rete un nodo non foglia (questo viene fatto internamente dalla classe), e quindi non c’è modo di inserire un rack di tipo diverso da InnerNode. Dunque concludo qui l’analisi della classe, non potendo migliorare ulteriormente la copertura sui metodi analizzati.

**Risorse**

**Category partition per Zookeeper:**

**create:**

Il parametro path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particolare il metodo ha un comportamento differente a seconda dei vari valori del parametro path: Se il path coincide con un nodo già esistente, allora viene sollevata una eccezione di tipo KeeperException.NodeExists; Se il path è mal formato viene sollevata una eccezione del tipo IllegalArgumentException; Se il path presenta come genitore un nodo ephemeral viene sollevata una eccezione del tipo KeeperException.NoChildrenForEphemeralNodes; Se il genitore presente nel path non esiste viene sollevata una eccezione del tipo KeeperException.NoNode; Se il path coincide con un nodo già presente viene sollevata una eccezione del tipo KeeperException.NodeExists. Il parametro data corrisponde ai dati iniziali del nodo e possiamo considerare come valori per il test {[],null,“1010”.getBytes()}.La lista di acl sono le acl che verranno inserite nel nodo, trattandosi di una lista ha senso prevedere : {vuota,null,List.of(acl)} con un acl qualsiasi.La createMode specifica se il nodo debba essere ephemeral o persistent e può assumere i valori {CreateMode.PERSISTENT,CreateMode.PERSISTENT\_SEQUENTIAL,CreateMode.EPHEMERAL,CreateMode.EPHEMERAL\_SEQUENTIAL,CreateMode.CONTAINER,CreateMode.PERSISTENT\_SEQUENTIAL\_WITH\_TTL}.

#### delete:

Il parametro path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particolare il metodo ha un comportamento differente a seconda dei vari valori: Se il path è inesitente viene sollevata una eccezione del tipo KeeperException.NoNode; Se il path corrisponde ad un nodo con figli allora viene sollevata una eccezione del tipo; Se il path è malformato viene sollevata una eccezione del tipo IllegalArgumentException; Version è la versione del nodo da rimuovere, quindi mi aspetto che sia un numero intero positivo. Il metodo da specifica fornisce una eccezione se la versione non coincide con la versione del nodo (se esiste), quindi ha senso prevedere due possibili valori per version che sono: {versione corretta, versione non corretta}, in realtà analizzando bene la documentazione si nota che il valore -1 viene inteso come “tutte le versioni vanno bene”, quindi ho tirato fuori questi 3 valori possibili {-2,-1,0} volendo intendere con 0 e -2 i casi “versione corretta”, “versione non corretta” e con il valore -1 il caso particolare.

#### exists:

path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particolare il metodo ha un comportamento differente a seconda dei vari valori: Se il path è inesistente viene ritornato null; Se il path è malformato viene sollevata una eccezione del tipo IllegalArgumentException.watch è boolean prende valori {true,false} e indica se voglio osservare o no

getACL: path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particoalre il metodo ha un comportamento differente a seconda dei vari valori: Se il path è inesistente viene sollevata una eccezione del tipo KeeperException.NoNode Stat viene riempita come parametro di output quindi no significativa nell’analisi.

getChildren: path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particolare il metodo ha un comportamento differente a seconda dei vari valori: Se il path è inesistente viene sollevata una eccezione del tipo KeeperException.NoNode watch è boolean e può assumere i seguenti valori {true,false}

getData: stat è parametro di output,path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particolare il metodo ha un comportamento differente a seconda dei vari valori del parametro path: Se il path è inesistente viene sollevata una eccezione del tipo KeeperException.NoNode. watch può assumere i valori {true,false}

setData: path può assumere tutti i valori riportati nell’enumerazione qui sotto, in particolare il metodo ha un comportamento differente a seconda dei vari valori: Se il path è inesistente viene sollevata una eccezione del tipo KeeperException.NoNode; data sono i dati quindi può asusmere gli stessi valori identificati in precedenza: {null,[],“1010”.getBytes()} version è la versione e segue gli stessi casi identificati in precedenza ovvero {-1,0,2}

Dominio per il parametro String path:

1. "“;
2. Path ben formato (ovvero che usa gli”/" anziché “"), ad esempio”/this/is/a/valid/path“;
3. Path mal formato, ad esempio”\this\is\not/a/valid/path";
4. null;
5. Path esistente (ovvero il nodo a cui fa riferimento esiste già);
6. Path inesistente (il nodo a cui fa riferimento non esiste);
7. Nodo con figli (ovvero il nodo a cui fa riferimento il path ha dei sottonodi);
8. Nodo ephemeral (corrispondente ad un nodo di tipo Ephemeral

### Casi di test selezionati Zookeeper:

#### prima iterazione

Tabella 1: Casi di test per la prima iterazione ZookKeeper

|  |  |  |
| --- | --- | --- |
| method | params | expected |
| create | “/test”,“1010”.getBytes(),List.of(acl),CreateMode.PERSISTENT | success |
| create | “/test”,“1010”.getBytes(),vuota,CreateMode.EPHEMERAL | exception Bad acl |
| create | “/test”,“1010”.getBytes(),null,CreateMode.EPHEMERAL | exception bad acl |
| create | “/test”,[],vuota,CreateMode.PERSISTENT | exception bad acl |
| create | “/”,“1010”.getBytes(),List.of(acl),CreateMode.PERSISTENT | exception node exists |
| create | “/test”,[],null,CreateMode.PERSISTENT | exception bad acl |
| create | “/test”,[],List.of(acl) | CreateMode.PERSISTENT\_SEQUENTIAL |
| create | “/testfail/no/parent”,[],List.of(acl),CreateMode.PERSISTENT | exception NoNodeException |
| create | “/testfail/no2/parent”,“1010”.getBytes(),List.of(acl),CreateMode.PERSISTENT | exception NoNodeException |
| create | “\kestfail/\ladpath”,[],List.of(acl),CreateMode.PERSISTENT | exception IllegalArgumentException |
| create | “\kestfail/\ladpath”,[],List.of(acl),CreateMode.PERSISTENT | exception IllegalArgumentException |
| create | “/provaeph”,“1010”.getBytes(),List.of(acl),CreateMode.PERSISTENT | exception NoChildrenForEphemeralsException (qui intendo creare questo nodo ponendo come parent un nodo ephemeral) |
| create | “/provaeph”,[],List.of(acl),CreateMode.PERSISTENT | exception NoChildrenForEphemeralsException (qui intendo creare questo nodo ponendo come parent un nodo ephemeral) |
| create | "",[],List.of(acl),CreateMode.PERSISTENT | exception illegalArgumentException (stringa vuota non è un path valido) |
| create | "“,”1010".getBytes(),List.of(acl),CreateMode.PERSISTENT | exception IllegalArgumentException |
| delete | “/test\_no\_node”,0 | exception NoNodeException |
| delete | “/test\_bad\_version”,-2 | exception BadVersionException |
| delete | “/test\_bad\_version”,2 | exception BadVersionException (ogni test è eseguito in maniera isolata, quindi il nodo richiesto è appena creato e ha versione 0) |
| delete | “/test\_delete”,0 | success |
| delete | “/test\_delete”,-1 | success |
| exists | “/”,false | success |
| exists | “/test\_exists”,false(watcher default registrato) | success |
| exists | “/test\_exists”,true(watcher default registrato) | success |
| exists | “/test\_exists”,true(watcher default non registrato) | exception IllegalStateException |
| setData | “/test”,“1010”.getBytes(),-1 | success |
| setData | “/test”,[],0 | success |
| setData | “/test”,[],1 | exception BadVersionException |
| setData | “/test\_no\_node”,“1010”.getBytes(),0 | exception NoNodeException(il nodo richiesto non è presente nel sistema) |
| setData | “/”,true | success |
| getData | “/test”,false | success |
| getData | “/test”,true | success |
| getData | “/test\_getnonode”,true | exception NoNodeException (il nodo richiesto non è presente nel sistema) |
| getACL | “/test” | success |
| getACL | “/test\_no\_node” | exception NoNodeException |
| setACL | “/test”,List.of(acl),-1 | success |
| setACL | “/test”,List.of(acl),10 | exception BadVersionException |
| setACL | “/test”,[],-1 | success |
| getChildren | “/test”,false | success |
| getChildren | “/test”,true | success |

#### Seconda iterazione

|  |  |  |
| --- | --- | --- |
| method | params | expected |
| addWatch | “/testAdd”,AddWatchMode.PERSISTENT | success |
| addWatch | “/testAdd”,AddWatchMode.PERSISTENT\_SEQUENTIAL | success |
| addWatch(versione con 3 parametri) | “/testAdd”,new MyWatcher(), AddWatchMode.PERSISTENT | success |
| addWatch(versione con 3 parametri) | “/testAdd”,new MyWatcher(), AddWatchMode.PERSISTENT\_SEQUENTIAL | success |
| addWatch(versione con 3 parametri) | “/testAdd”,null, AddWatchMode.PERSISTENT | Exception IllegalArgumentException |
| removeWatches | “/testRemove”,new WatcherToRemove(), Watcher.WatcherType.Any,false | success |
| removeWatches | “/testRemove”,new WatcherToRemove(), Watcher.WatcherType.Any,true | success |
| removeWatches | “/testRemove”,new WatcherToRemove(), Watcher.WatcherType.Data,false | success |
| removeWatches | “/testRemove”,new WatcherToRemove(), Watcher.WatcherType.Data,true | success |
| removeWatches | “/testRemove”,new WatcherToRemove(), Watcher.WatcherType.Children,false | success |
| removeWatches | “/testRemove”,new WatcherToRemove(), Watcher.WatcherType.Children,true | success |
| removeWatches | “/testRemove”,new WatcherToRemove() (il watcher non viene prima registrato), Watcher.WatcherType.Any,false | Exception NoWatcherException |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2), Watcher.WatcherType.Any,false | success |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2), Watcher.WatcherType.Any,true | success |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2), Watcher.WatcherType.Data,false | success |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2), Watcher.WatcherType.Data,true | success |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2), Watcher.WatcherType.Children,false | success |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2), Watcher.WatcherType.Children,true | success |
| removeAllWatches | “/testRemove”,List.of(watch1,watch2) (i watcher non sono collegati), Watcher.WatcherType.Any,false | exception NoWatcherException |
| getEphemerals | no params | success |
| getEphemerals | “/” | success (legge tutti i nodi ephemeral) |
| getEphemerals | “/test\_eph\_from\_prefix” | success (legge tutti i nodi ephemeral a partire dal path) |