Università degli Studi di Roma Tor Vergata

Testing Open-Source Projects

Autore:

Leonardo Beniamino Petraglia

January 26, 2021

Contents

Chapter 1

Introduzione

Lo scopo del progetto è stato quello di effettuare attività di testing su due progetti open-source Apache, nel particolare sono stati presi in esame Bookkeeper e Zookeeper. Tali attività di testing sono state cosi strutturate:

- Individuazione di alcune classi per ciascuno dei 2 progetti;
- Individuazione delle category partition;
- Composizione di una suite minimale per ogni metodo pubblico delle classi
- Implementazione dei casi di test
- Misura dell'aguatezza dei test tramite Jacoco
- Killing dei mutanti individuati da PIT

Il lavoro ha avuto come fine quindi quello di acquisire e fornire un sufficiente livello di fiducia sul funzionamento delle classi prese in esame tramite l'esecuzione di esperimenti condotti in un ambiente controllato.
Per eseguire queste attività si è effettuato il fork dei progetti suddetti dalle rispettive repository Github; successivamente sono state eliminate le classi di test già presenti all'interno di essi per sostituirle con quelle create. Entrambi i progetti sono poi stati collegati con *Travis-CI* e *Sonar-Cloud* configurando opportunamente i file *travis.yml* e *sonar-project.properties* per adattarli alle attività di Automation & Continuous Testing. Il primo servizio si occupa infatti di eseguire la build del progetto e i casi di test, il secondo va invece ad eseguire controlli qualitativi. Sono stati poi configurati i *pom.xml* dei moduli per abilitare la misurazione della line-coverage di Jacoco e l'individuazione dei mutanti di Pitest.

Chapter 2

Individuazione delle Classi

2.1 Bookkeeper

Per quanto riguarda Bookkeeper sono state prese in considerazione le classi:

- **BufferedChannel**, essa offre un servizio di ottimizzazione per quanto riguarda la scrittura dei Ledger di Bookkeeper su un Log File. Un Ledger è l'unità di base di storage di Bookkeeper e consiste in una sequenza di Log entries. Tale classe permette di mantenere in memoria un certo ammontare di Ledger per poter effettuare la scrittura di tutti quanti una sola volta piuttosto che effettuare molteplici scritture, attività tipicamente dispendiose in termini di tempo di esecuzione;

- **BookieStatus**, essa va a determinare lo stato di un Bookie ovvero uno storage server di Bookeeper che contiene Ledger e comunica in modo distribuito. Lo stato di un Bookie può essere READ_ONLY ovvero disponibile in sola lettura e READ_WRITE disponibile sia in lettura sia in scrittura.

2.2 Zookeeper

- **ByteBufferInputStream**, è una estensione della classe astratta InputStream che è a sua volta una implementazione dell'interfaccia Closable. Fornisce dei metodi per operare su un Bytebuffer, tra cui la lettura da esso;

- **ByteBufferOutputStream**, come nella classe precedente si tratta di una estensione della classe astratta OutputStream che è a sua volta un'implementazione dell'interfaccia

Closable e Flushable. Fornisce dei metodi per operare su un ByteBuffer e in particolare la scrittura su di esso;

- **IOUtils**, fornisce delle facility riguardo oggetti Closable, in particolare permette di chiudere degli stream oppure di copiare dei byte tra un InputStream ad un OutputStream;
- **StringUtils**, fornisce un metodo per la divisione di una stringa secondo un separatore scelto e un metodo per unire una lista di stringhe secondo un delimitatore.

Chapter 3

Category Partition

La creazione delle partizioni è stata effettuata tramite la procedura descritta durante il corso, vengono pertanto considerati sia il dominio di input sia ulteriori variabili considerate influenti per quanto riguarda i casi di test dei metodi pubblici esposti.

3.1 Bookkeeper

3.1.1 BufferedChannel

I metodi pubblici presenti nella classe sono:

- **public void write(Bytebuf src)**, esso va a scrivere i dati presenti nel ByteBuf preso come sorgente sul FileChannel associato al BufferedChannel. Essendo un oggetto complesso è opportuno andare a considerare un'istanza corretta e una non corretta. Considerato poi che all'interno del metodo viene richiamato il metodo *readableBytes()* si potrebbe andare a considerare sia un'istanza con una lunghezza pari a zero sia un'istanza con una lunghezza maggiore di 0. Si individuano pertanto le seguenti 3 istanze:
  - Istanza null
  - Istanza con len = 0
  - Istanza con len > 0

  Notiamo inoltre che all'interno del metodo sono prese in considerazione anche le due costanti *unpersistedBytes* e *unpersistedBytesBound* che indicano rispettivamente il numero di byte non ancora resi persistenti, ovvero non ancora scritti sul FileChannel, e il

limite al numero di byte che possono rimanere non persistenti. Al superamento di tale limite verrà forzata la scrittura sul file tramite la chiamata della funzione *flush()*. I valori che può assumere sono quindi:

- unpersistedBytesBound = 0
- unpersistedBytesBound = 0

La suite minimale individuata pertanto risulta:

- ByteBuf = null, unpersistedBytesBound = 0
- ByteBuf = Unpooled.buffer(0), unpersistedBytesBound = 1
- ByteBuf = Unpooled.buffer(1), unpersistedBytesBound = 1

- **public int read(ByteBuf dest, long pos, int length)**, esso va a leggere i byte dal FileChannel e li scrive in un ByteBuf di destinazione. Il parametro pos indica la posizione a partire dal quale effettuare la lettura mentre il parametro length indica la lunghezza per il quale scrivere; infine è necessario considerare la costante *writeCapacity* che va a indicare la capacità del buffer in cui scrivere. Di seguito le partizioni dei parametri suddetti:

  - writeCapacity <=0, WriteCapacity > 0
  - ByteBuf = null, ByteBuff con len = 0, ByteBuff con len > 0
  - pos <= 0, pos > 0
  - length <= 0, length > 0

  La suite minimale individuata è la seguente:

  - writeCapacity = 0, ByteBuf = null, pos = 0, length = 0
  - writeCapacity = 1, ByteBuf = Unpooled.buffer(0), pos = 1, length = 1
  - writeCapacity = 1, ByteBuf = Unpooled.buffer(1), pos = 1, length = 1

3.1.2 BookieStatus

I metodi pubblici presenti nella classe sono:

- **public BookieStatus parse(BufferedReader reader)**, esso si occupa di effettuare il parsing di un BookieStatus ritornando lo stesso in caso di successo, null in caso di errore.

L'unico parametro presente è BufferedReader che costituisce un oggetto complesso, lo partizioniamo quindi come segue:

  - Istanza null
  - Istanza corretta con reader non corretto
  - Istanza corretta con reader corretto

  Il BufferedReader può essere istanziato tramite un'istanza di StringReader che fornisce al reader una stringa del tipo: *LayoutVersion*, *BookieMode*, *LastUpdateTime*. La suite minimale può quindi essere la seguente:

  - reader = null
  - reader = "AnIncorrectString"
  - reader = layoutVersion + bookieMode + lastUpdateTime

- public void writeToDirectories(List<File> dir), esso va a scrivere in modalità best effort, lo status di un Bookie all'interno di molteplici directory individuate dal parametro dir, una lista di File. Essendo una lista un oggetto complesso individuiamo 3 possibili configurazioni:

  - Istanza null
  - Istanza corretta con numero di elementi pari a 0
  - Istanza corretta con numero di elementi maggiore di 0

  Una suite minimale pertanto può essere la seguente:

  - dir = null
  - dir = vuota
  - dir = File Corretto

- **public void readFromDirectories(List<File> dir)**, esso si occupa di leggere lo stato di un Bookie da uno dei file di stato forniti come input dal parametro dir. Se la lettura è portata a termine viene aggiornato lo stato del bookie. Se uno dei file di stato non è leggibile oppure non viene trovato esso verrà saltato e si passerà alla lettura del file successivo. Come nel caso precedente dir è un oggetto complesso pertanto individuiamo le seguenti configurazioni:

- Istanza null
- Istanza corretta con numero di elementi pari a 0
- Istanza corretta con numero di elementi > 0

Una possibile suite minimale è pertanto la seguente:

- dir = null
- dir = vuota
- dir = File corretto

3.2 Zookeeper

3.2.1 ByteBufferInputStream

- **public int read()**, il metodo va a leggere dal bytebuffer e va a restituire la posizione corrente dello stesso, in particolare restituisce gli 8 bit meno significativi. Dato che si va a leggere dal bytebuffer consideriamo questo come un parametro e andiamo a individuare le partizioni per esso considerando che si tratta di oggetto complesso:
  - Istanza null
  - Istanza corretta con lunghezza 0
  - Istanza corretta con lunghezza > 0

  Pertanto la suite minimale è la seguente:
  - null
  - ByteBuffer.allocate(0)
  - ByteBuffer.allocate(1)
- **public int available()**, il metodo restituisce il numero di byte rimanenti all'interno del ByteBuffer. Come parametro consideriamo ByteBuffer e le partizioni sono le seguenti:
  - Istanza null
  - Istanza corretta con lunghezza 0

- Istanza corretta con lunghezza > 0

La suite minimale è quindi:

- null
- ByteBuffer.allocate(0)
- ByteBuffer.allocate(1)

- **public int read(byte[] b, int off, int len)**, esso va a trasferire i byte da un ByteBuffer all'interno di un array di byte. Per i parametri consideriamo le seguenti partizioni andando ad aggiungere come sopra il ByteBuffer:
  - ByteBuffer: null, istanza corretta con lunghezza 0, istanza corretta con lunghezza $>$ 0
  - byte[] b:istanza null, len $= 0$, len > 0
  - int off: $<=0$, > 0
  - len $<= 0$; > 0

  Pertanto è stata individuata la seguente suite minimale:
  - null, null, 0, 0
  - ByteBuffer.allocate(0), new byte[0], 1, 1
  - ByteBuffer.allocate(1), new byte[1], 1, 1

- **public int read(byte[] b)**, il metodo fornisce la stessa funzionalità del metodo precendente ma preimposta il valore della posizione a 0 e la lunghezza per cui leggere alla totalità della lunghezza dell'array di byte passato come parametro. Consideriamo le partizioni come di seguito:
  - Istanza null
  - Istanza con lunghezza pari a 0
  - Istanza con lunghezza maggiore di 0

  La suite minimale pertanto risulta:
  - null

  - new byte[0]
  - new byte[1]

- **public long skip(long n)**, il metodo prende come parametro un long e va a spostare la posizione da cui leggere al numero preso come parametro. Consideriamo come i casi precedenti anche ByteBuffer:

  - ByteBuffer: null, istanza con lunghezza 0, istanza con lunghezza > 0
  - long n: $<=0$, > 0

  Quindi una suite minimale è la seguetne:

  - null, 0
  - ByteBuffer.allocate(0), 1
  - ByteBuffer.allocate(1), 1

- **public static void byteBuffer2Record(Bytebuffer bb, Record record)**, il metodo va a deserializzare il Bytebuffer passato come parametro per trasformarlo in un Record. Entrambi sono oggetti complessi pertanto individuiamo le seguenti partizioni:

  - ByteBuffer: null, lunghezza $= 0$, lunghezza > 0
  - Record: null, Istanza corretta

  Per istanziare correttamente il record utilizziamo uno dei costruttori usati all'interno del progetto per implementare l'interfaccia Record.

  La suite minimale è pertanto:

  - null, null
  - ByteBuffer.allocate(0), new CreateTxn()
  - ByteBuffer.allocate(100), new CreateTxn()

3.2.2 ByteBufferOutputStream

- **public void write(int b)**, il metodo va a scrivere all'interno di un ByteBuffer l'intero preso come parametro effettuando il casting a byte. Consideriamo come parametro oltre all'intero b anche il ByteBuffer pertanto le partizioni sono:

- ByteBuffer: null, lunghezza = 0, lunghezza > 0
- int b: <=0, >0

Da cui troviamo la suite minimale:

- null, 0
- ByteBuffer.allocate(0), 0
- ByteBuffer.allocate(1), 1

- **public void write(byte[] b)**, esso va a scrivere all'interno di un ByteBuffer l'array di byte passato come parametro. Come prima consideriamo oltre all'array di byte anche il ByteBuffer come parametro.
  - ByteBuffer: null, lunghezza = 0, lunghezza > 0
  - byte[] b: null, lunghezza = 0, lunghezza > 0

  La suite minimale pertanto è:
  - null, null
  - ByteBuffer.allocate(0), new byte[0],
  - ByteBuffer.allocate(1), new byte[1]

- **public void write(byte[] b, int off, int len)**, il metodo va a scrivere all'interno di un ByteBuffer l'array di byte passato come parametro a partire da un offset e per una determinata lunghezza. Come prima consideriamo il ByteBuffer in aggiunta ai parametri di input del metodo:
  - ByteBuffer: null, lunghezza = 0, lunghezza > 0
  - byte[] b: null, lunghezza = 0, lunghezza > 0
  - int off: <= 0, > 0
  - int len: <= 0, > 0

  La suite minimale è quindi:
  - null, null, 0, 0

- ByteBuffer.allocate(0), new byte[0], 1, 1
- ByteBuffer.allocate(1), new byte[1], 0, 1

- **public static void record2ByteBuffer(Record record, ByteBuffer bb)**, il metodo va a serializzare il record passato come parametro per trasformarlo in un ByteBuffer. Entrambi sono oggetti complessi pertanto individuiamo le seguenti partizioni:

  - ByteBuffer: Istanza null, lunghezza = 0, lunghezza > 0
  - Record: Istanza null, Istanza corretta

  Pertanto la suite minimale è la seguente:

  - null, null
  - ByteBuffer.allocate(0), new CreateTxn()
  - ByteBuffer.allocate(100), new CreateTxn()

3.2.3 IOUtils

- **public static void closeStream(Closeable stream)**, tale metodo va a chiamare il metodo *cleanup* preimpostando uno dei suoi parametri a null e ricevendo come unico parametro uno stream di tipo Closeable. Le partizioni per tale parametro sono le seguenti:

  - Istanza nulla, Istanza corretta

  La suite minimale risultante è quindi:

  - null
  - new OutputStream()

- **public static void cleanup(Logger log, Closeable... closeables)**, il metodo va a cericare di chiudere uno o più stream associati a dei file passati come parametri con eventualmente un log. Con i 2 parametri suddetti scriviamo le partizioni come segue:

  - Logger log: Istanza null, Istanza corretta
  - Closeable: Istanza null, Array con lunghezza 0, Array con lunghezza > 0

La suite minimale che ne segue è:

- null, null
- log, new OutputStream[]
- log, new OutputStream[]new OutputStream()

- **public static void copyBytes(InputStream in, OutputStream out, int buffSize, boolean close)**, il metodo va a copiare i byte contenuti nello stream sorgente individuato da *Inpustream in* nella destinazione individuata da *Outpustream out* attraverso un array di byte la cui lunghezza è individuata da *buffSize*. L'ultimo parametro va a indicare se al termine della copia dei byte andare a chiudere o meno gli stream precedenti. Le partizioni per tale dominio di input risultano:

  - InputStream: Istanza nulla, Istanza corretta
  - OutputStream: Istanza nulla, Istanza corretta
  - int buffSize: $<= 0$, > 0
  - boolean close: false, true

  La suite minimale risultante pertanto è:

  - null, null, 0, false
  - new InputStream(), new OutputStream(), 1, true

- **public static void copyBytes(InputStream in, OutputStream out, int buffSize)**, esso va a copiare i bytes da un Inputstream sorgente ad un OutputStream destinazione specificando la dimensione del buffer usato come tramite per tale trasferimento dati. La partizione individuata risulta:

  - InputStream: Istanza nulla, Istanza corretta
  - OutputStream: Istanza nulla, Istanza corretta
  - int buffSize: $<= 0$, > 0

  La suite minimale è pertanto:

  - null, null, 0
  - new InputStream(), new OutputStream, 1

3.2.4 StringUtils

- **public static List<String> split(String values, String separator)**, il metodo ritorna una lista di stringhe costruita a partire da una stringa presa come parametro e che viene suddivisa in base ad un separatore. Possiamo considerare le stringhe come un oggetto complesso pertanto applichiamo le stesse regole applicate precedentemente.

  - String value: Istanza null, Istanza con lunghezza = 0, Istanza con lunghezza > 0
  - String separator: Istanza null, Istanza con lunghezza = 0, Istanza con lunghezza > 0

  La suite minimale è la seguente:

  - null, null
  - "", ""
  - "s", "st"

- **public static String joinStrings(List<String> list, String delim)**, tale metodo prende in input una Lista di stringhe e va ad unire tali stringhe in un'unica sola tramite il delimitatore specificato come secondo parametro.

  - List<String> list: Istanza null, Istanza con numero di stringhe pari a 0, Istanza con numero di stringhe > 0
  - String delim: Istanza null, Istanza con lunghezza = 0, Istanza con lunghezza > 0

  La suite minimale è pertanto:

  - null, null
  - new ArrayList<>(), ""
  - new ArrayList<>().add("string"), "s"

Chapter 4

Implementazione Casi di Test

L'implementazione dei casi di test è avvenuta tramite il supporto del framework JUnit. Per l'esecuzione dei casi di test relativi alle category partition si è fatto uso della notazione @RunWith(Parameterized.class), essa infatti permette una scrittura e configurazione semplice ed espandibile. Per la preparazione dell'ambiente di test cosi come per il suo ripristino allo stato iniziale sono state usate rispettivamente le annotazioni @Before/@BeforeClass e @After/@AfterClass. Infine per poter scrivere un'unica classe di test per ogni classe presa in esame è stata utlizzata la notazione @RunWith(Enclosed.class).

4.1 Preparazione dell'ambiente di test

4.1.1 Bookkeeper

BufferedChannel Per tale classe è stata creata un'unica classe di test annotata con @RunWith(Enclosed.test) all'interno del quale sono state definite ulteriori classi: una per le operazioni di scrittura e una per le operazioni di lettura, entrambe annotate con @RunWith(Parameterized.class) Sono stati usati alcuni metodi addizionali per la preparazione dell'ambiente di test e per il ripristino dello stesso alla fine dell'esecuzione dei test:

- **close()**, tale metodo è annotato con @AfterClasse pertanto viene eseguito a valle dell'esecuzione di ogni caso di test. Esso va a chiudere il bufferedChannel.
- **openChannel**, viene eseguito con le stesse modalità del metodo precedente e va ad aprire un canale relativo ad un file di test.

BookieStatus Come nella classe precedente è stata creata un'unica classe di test annotata con @RunWith(Enclosed.class) con all'interno una classe per le operazioni di lettura e scrittura e una per l'operazione di parsing. Sono stati scritti inoltre due metodi ausiliari:

- **createDirectory()**, annotato con @BeforeClass pertanto viene eseguito una sola volta prima dell'esecuzione dei casi di test; esso va a creare un file temporaneo su cui operare con i casi di test
- **deleteDirectory()**, annotato con @AfterClass pertanto viene eseguito una sola volta a valle dell'esecuzione di tutti i casi di test; esso va a eliminare il file temporaneo precedentemente creato

4.1.2 Zookeeper

Per quanto riguarda questo progetto le classi sotto test e in particolare i casi di test derivanti dalle category partition sono stati eseguiti senza l'ausilio di funzioni di preparazione dell'ambiente di test. Unica eccezione è il l'utilizzo del metodo **closeFiles** annotato come @After all'interno della classe **IOUtilsTest** per andare a chiudere i file associati ai vari stream utilizzati.

4.2 Line Coverage con Jacoco

L'utilizzo del plugin Jacoco ha permesso di aumentare il livello di line coverage delle applicazioni sotto test. Questo ha comportato la creazione e l'esecuzione di test addizionali rispetto a quelli individuati tramite category partition e che sono stati inseriti o come ulteriore set di parametri in aggiunta a quelli preesistenti oppure all'interno di classi e metodi appositi. Per configurarlo, considerando ad esempio Zookeeper, è stato necessario aggiungere all'interno del file *pom.xml* del modulo principale con la seguente specifica che va a indicare dove posizionare il report una volta generato:

```
<aggregate.report.dir>moduleTestZookeeper/target/site/
jacoco-aggregate/jacoco.xml</aggregate.report.dir>
```

Per abilitare invece il plugin e collegarlo a SonarCloud si è inserito la seguente proprietà nel file *pom.xml* del modulo contenente le classi prese in esame:

```
    <properties>
    <sonar.coverage.jacoco.xmlReportPaths>${project.basedir}
    /../${aggregate.report.dir}</sonar.coverage.jacoco.xmlReportPaths>
  </properties>
```

Infine all'interno del modulo ausiliario creato appositamente (*moduleTestZookeeper*)è stata inserita la seguente dipendenza per permettere di eseguire l'analisi del codice e la generazione del report nella fase *Verify* di Maven:

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.jacoco</groupId>
                <artifactId>jacoco-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <id>report</id>
                        <goals>
                            <goal>report-aggregate</goal>
                        </goals>
                        <phase>verify</phase>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

4.3 PIT Mutation

L'utilizzo del software PIT ha permesso di valutare l'adeguatezza dei test prima configurati rispetto alle mutazioni create appositamente sul codice testato. Similmente a Jacoco è stato

necessario configurare appositamente i file *pom.xml* del progetto, in particolare nel modulo contenente le classi di test abbiamo

```
<plugin>
        <groupId>org.pitest</groupId>
        <artifactId>pitest-maven</artifactId>
        <version>1.5.2</version>
        <configuration>
          <targetClasses>
            <param>org.apache.zookeeper.common.IOUtils</param>
            <param>org.apache.zookeeper.server.ByteBufferInputStream</param>
            <param>org.apache.zookeeper.server.ByteBufferOutputStream</param>
            <param>org.apache.zookeeper.common.StringUtils</param>
          </targetClasses>
          <targetTests>
            <param>org.apache.zookeeper.IOUtilsTest</param>
            <param>org.apache.zookeeper.ByteBufferInputStreamTest</param>
            <param>org.apache.zookeeper.ByteBufferOutputStreamTest</param>
            <param>org.apache.zookeeper.StringUtilsTest</param>
          </targetTests>
        </configuration>
</plugin>
```

Per ognuna delle classi prese in esame viene pertanto generato un report contenente il numero di mutanti generati e una percentuale di quelli che sono stati *KILLED*, ovvero coperti dai casi di test. Per raggiungere la percentuale più possibilmente alta di mutazioni KILLED si è fatto inoltre uso del framework Mockito e in alcuni casi dell'estensione PowerMockito, essi permettono infatti di simulare alcuni comportamenti riguardanti metodi o istanze delle classi testate.

4.3.1 Bookkeeper

BufferedChannel Il report di PIT segnala la creazione di 60 mutanti di cui 42 sono stati eliminati. Essi sono stati eliminati o tramite la category partition precedentemente illustrata o tramite la creazione di casi di test ad hoc.

BookieStatus Il report di PIT segnala la creazione di 27 mutanti di cui 18 vengono eliminati, come nel caso precedente l'eliminazione è stata effettuata sia con casi di test derivanti dalla category partition sia da casi di test creati ad hoc per la loro eliminazione. Si riportano di seguito alcuni commenti:

- **riga 113**, in riferimento al metodo *readFromDirectories* nonostante vengano coperti entrambi i branch tramite i casi di test, andando a scrivere sul *log* non è stato possibile differenziare i casi di test ed eliminare la mutazione
- **riga 164**, in riferimento al metodo *writeToDirectories* come nel caso precendente nonostante vengano coperti entrambi i branche tramite i casi di test, andando a scrivere sul *log* non è stato possibile differenziare i casi di test ed eliminare la mutazione.

4.3.2 Zookeeper

IOUtils Il report di PIT segnala la creazione per questa classe di 17 mutanti di cui 12 risultano eliminati o dai test derivanti dalla category partition o da quelli appositamente aggiunti. Riportiamo di seguito alcuni commenti per tale classe:

- **righe 57-59**, per l'eliminazione della mutazione è stato fatto uso di Mockito per far lanciare una IOException alla chiamata del metodo close.
- **riga 86**, per l'eliminazione di tale mutante si è fatto ricorso a Mockito, nel particolare si è simulato il comportamento della classe OutputStream all'esecuzione del metodo close facendo partire un'eccezione IOException;
- **riga 88**, come per il caso precedente si è utilizzato Mockito per lanciare una IOException alla chiamata del metodo close riguardante l'Inputstream. Viene inoltre simulato il comportamento del metodo *read()* in quanto essendo l'Inputstream privo di dati avrebbe causato un'esecuzione infinita.

- **riga 84**, la mutazione risulta eliminata come conseguenza dell'eliminazione delle 2 precedenti mutazioni.
- **righe 92-93-94**, le mutazioni non risultano eliminate in quanto risulta difficile simulare un comportamento similmente ai casi precedenti essendo *closeStream* un metodo statico all'interno di un ulteriore metodo statico, nonostante l'utilizzo di PowerMockito;
- **riga 113**, la mutazione non viene eliminata in quanto per poterlo fare sarebbe necessario far ritornare al metodo *read* il valore 0, il che però risulta in un loop infinito di esecuzione
- **riga 115**, per eliminare la mutazione è stato utilizzato PowerMockito per modificare la variabile locale *PrintStream* e non farla risultare come null.

StringUtils Il report segnala la creazione di 8 mutanti di cui risultano eliminati tutti quanti. Data anche la natura della classe non si è fatto uso di Mockito per simulare ulteriori comportamenti, i mutanti risultano eliminati tramite la suite minimale precedentemente decritta insieme a due ulteriori casi di test.

ByteBufferInputStream Dal report di PIT notiamo come di 16 mutazioni totali ne vengano eliminate 13, esse sono state eliminate tramite la suite minimale precedentemente individuata e con l'aggiunta di ulteriori test per l'aumento della coverage.

ByteBufferOutputStream Il report segnala un'unica mutazione che viene eliminata dalla suite minimale individuata per i metodi pubblici della classe.

Chapter 5

Links

Vengono riportati di seguito i link di GitHub, Travis-CI e SonarCloud dei progetti presi in esame:

Bookkeeper

- GitHub: https://github.com/leopetr95/bookkeeper
- Travis-CI: https://travis-ci.com/github/leopetr95/bookkeeper
- SonarCloud: https://sonarcloud.io/dashboard?id=leopetr95_bookkeeper

Zookeeper

- GitHub: https://github.com/leopetr95/zookeeper
- Travis-CI: https://travis-ci.com/github/leopetr95/zookeeper
- SonarCloud: https://sonarcloud.io/dashboard?id=leopetr95_zookeeper