## Introduzione

L'obiettivo di questo studio è stato applicare le tecniche e gli strumenti visti nel modulo del corso relativo il *software testing* a due progetti open source, *Bookkeeper* ed *AVRO.* Lo studio si è concentrato su due classi Java per ogni singolo progetto, individuate con la metodologia descritta nel paragrafo *Individuazione classi.* Su ogni singola classe sono stati eseguiti esperimenti in un ambiente controllato al fine di poter acquisire sufficiente fiducia sul loro funzionamento.

## Configurazione ambiente di lavoro

Per la configurazione dell'ambiente di lavoro sono stati effettuati i fork dei due progetti sulla repository GitHub salvatorefoderaro/avro e salvatorefoderaro/bookkeeper. Sono stati rimossi i test presenti di default, rimossi i profili dai file *pom.xml* dei due progetti. Inoltre è stata effettuata l'integrazione con **Travis-CI** e **SonarCloud**, tramite la configurazione dei file *.travis.yml* e *sonar-projects.properties*.

## Individuazione classi

Per l'individuazione delle classi da testare sono stati tenuti presenti i risultati delle attività svolte con il Prof. Falessi. Sono state apportate alcune modifiche affinché venissero calcolate le metriche per l'ultima versione rilasciata di ogni progetto. Per la scelta delle classi da testare si è tenuto conto della metrica *numberRevisions*, che indica il numero di volte che il file appare in un commit su GitHub. Questo ha portato alla scelta delle seguenti classi per i due progetti.

- **BookKeeper**
  - *BufferedReader:* tra gli elementi base di Bookkeeper sono presenti i *ledger*. Un *ledger* è una sequenza di *entry*, ognuno dei quali è composto da una sequenza di dati. Quando un server riceve una entry da parte dei client per essere scritta, l'entry viene aggiunta all'*entry log*. Per motivi di performance, l'entry log bufferizza in memoria le varie entry e le scrive in modo asincrono sul file di log. Per la bufferizzazione viene utilizzata la classe *BufferedChannel;* offre un layer bufferizzato per l'esecuzione delle operazioni su un *file channel.* La singola entry è composta dai campi:
    - long ledgerID
    - long entryID
    - long LAC
    - long length
    - byte[] data
  - *DigestManager:* per l'invio delle entry al server, il client utilizza la classe per il calcolo del digest da aggiungere al pacchetto da inviare, mentre il server la utilizza per verificare che ila correttezza del pacchetto ricevuto.
- **AVRO**
  - *BinaryData:* AVRO è un sistema per la serializzazione dei dati basato su *schemi*. Vengono utilizzati per definire in che che modo il flusso di dati deve essere scritto, o il modo in cui deve essere letto. Durante l'esecuzione delle due operazioni, lo *schema* è sempre presente. La classe fornisce degli strumenti per la manipolazione di dati binari.
  - *SpecificData:* la classe offre delle utility per la generazione e la gestione di oggetti di tipo *schema* per tipi di dato specifici del linguaggio Java.

## Category Partition ed analisi boundary-values

Per la stesura della *category partition è* stato utilizzato il procedimento illustrato a lezione, in base al tipo di dato del parametro di cui andare ad effettuare il partizionamento in categorie. Per i parametri composti sono stati considerati per l'analisi gli attributi dell'oggetto che vengono utilizzati nel flusso di esecuzione del metodo. Successivamente è stata eseguita l'analisi dei *boundary-values* per la stesura di una prima suite di test minimale.
Per il partizionamento degli oggetti complessi, come gli oggetti *schema* di **AVRO** o le *entry* di **Bookkeeper**, sono stati considerati gli attributi semplici che hanno portato alla costruzione degli oggetti stessi. Per questo motivo nell'analisi è presente una distinzione tra *nomeParametro.nomeAttributo* e *nomeAttributo;* il secondo è riferito all'istanza dell'oggetto di cui vogliamo andare a testare il metodo (*BufferedChannel, DigestManager).*

Per l'identificazione dei test è stato utilizzato il criterio unidimensionale: ogni test viene scelto per coprire tutte le classi di equivalenza considerate.

## Category Partition - Bookkeeper

### BufferedChannel - write (ByteBuf src)

Il metodo *write* scrive
il contenuto dell'oggetto *ByteBuf* sul *filechannel* associato all'oggetto *BufferedChannel*.

- *ByteBuf src.* Trattandosi di un oggetto complesso, per la category partition sono state considerati i casi in cui l'istanza sia nulla o valida. Per le istanze valide, trattandosi di un buffer di dati dotato di una determinata lunghezza, è stata considerata la lunghezza del buffer.
  - {null, src.len = 0, src.len > 0}
  - Analisy boundary-value: {null, src.len = 0, src.len =1}
- *long unpersistedBytesBound.* L'attributo rappresenta il numero di limite di byte non consistenti. Se il valore è maggiore di 0, viene effettuato il flush sul *file channel* dopo ogni scrittura. In caso contrario, il flush viene effettuato solamente al riempimento del buffer.
  - {<=0, >0}
  - Analisy boundary-value: {-1, 1}

La *suite di test minimale* identificata è la seguente:

- {null, -1}, {src.len = 0, 1, src.len = 1, 1}

### BufferedChannel - read (ByteBuf dest, long pos, int length)

Il metodo *read* scrive sul buffer di destinazione (*dest*) il contenuto del *filechannel* associato al BufferedChannel.

- *int capacity.* Il valore rappresenta la capacità di scrittura e lettura del buffer. Per un'istanza valida, sono state considerate due classi per il partizionamento:
  - {= 0, > 0}
  - Analisi boundary-value: {0, 1}
- *ByteBuf dest.* L'oggetto rappresenta il buffer di destinazione. E' stata considerata l'istanza non valida, e due tipologie di istanze valide che si caratterizzano per l'attributo *len*.
  - {null, dst.len = 0, dst.len > 1}
  - Analisi boundary-value: {null, dst.len = 0, dst.len = 1}
- *long pos.* Il valore rappresenta la posizione iniziale della scrittura. Per il partizionamento sono stati considerati tre casi in riferimento alla capacità di scrittura del *BufferedChannel.*
  - {< writeCapacity, = writeCapacity, > writeCapacity}
  - Analisi boundary-value: {writeCapacity - 1, writeCapacity, writeCapacity + 1}
- *int length.* Il valore rappresenta la lunghezza della scrittura. Dipende dalla capacità di scrittura del buffer, dalla posizione di scrittura ma anche dalla capacità del buffer di destinazione.
  - {<= writeCapacity - pos && <= dest.len, <=writeCapacity - pos && > dest.len, > writeCapacity - pos}
  - Analisi boundary-value: { (writeCapacity - pos) && (dest.len), (writeCapacity - pos) && (dest.len) + 1, (writeCapacity - pos) + 1}

La *suite di test minimale* identificata è la seguente

- {0, null, -1, 0}, {0, dst.len = 0, 0, 1}, {1, dst.len = 1, 2, 2}

### DigestManager - computeDigestAndPackageForSending (long entryId, long lastAddConfirmed, long length, ByteBuf data)

Il metodo permette al client di preparare l'entry da inviare al server, andando a calcolare il *digest* e concatenarlo con il buffer di dati a cui associare la *entry*.

- *DigestType* digestType. Il valore rappresenta i diversi algoritmi per il calcolo del *digest.* Trattandosi di una Enum, e' stata considerata una classe di equivalenza per ogni possibile valore:
  - {HMAC, CRC32, CRC32C, DUMMY}
  - Analisi boundary-value: {HMAC, CRC32, CRC32C, DUMMY}

- ByteBuf data. L'oggetto rappresenta il buffer di byte che rappresenta l'entry a cui verrà concatenato il digest prima dell'invio
  - {null, data.len = 0, data.len > 0}
  - Analisi boundary-value: {null, data.len = 0, data.len = 1}
- *long lastAddConfirmed.* Rappresenta il valore dell'ultima entry registrata e di cui il client ha ricevuto riscontro.
  - {<0, =0, >0}
  - Analisi boundary-value: {-1, 0, 1}
- *long entryId.* Rappresenta l'ID della singola entry che si vuole inviare al server. La categorizzazione è stata effettuata assumendo il vincolo che *l'entryID* di interesse non sia superiore all'ultima *entry* di cui il client ha avuto riscontro.
  - {<= lastAddConfirmed, > lastAddConfirmed}
  - Analisi boundary-value: {lastAddConfirmed, lastAddConfirmed + 1}
- *long length.* Lunghezza del buffer dei dati.
  - {=0, >0}
  - Analisi boundary-value: {0, 1}

La *suite di test minimale* identificata è la seguente

- {HMAC, null, -1, -1, -1}, {CRC32, data.len = 1, 1, 2, 1}, {CRC32C, data.len = 0, 0, 2, 0}, {DUMMY, data.len = 1, 1, 2, 1}

### DigestManager – verifyDigestAndReturnData (long entryId, ByteBuf dataReceived)

Il metodo permette al server, in ricezione, di verificare il digest dell'entry ricevuta dal client ed ottenere i dati, rimuovendo in questo modo l'header dovuto al *digest*.

- *DigestType digestType*. Trattandosi di una **enum**, e' stata considerata una classe di equivalenza per ogni possibile valore:
  - *{HMAC, CRC32, CRC32C, DUMMY}*
  - Analisi boundary-value: *{HMAC, CRC32, CRC32C, DUMMY}*
- *long ledgerID.* L'ID del ledger di appartenenza della singola entry.
  - {<0, =0, >0}
  - Analisi boundary-value: {-1, 0, 1}
- *long entryID.* L'ID della entry.
  - *{<0, = 0, >0}*
  - Analisi boundary-value: *{-1, 0, 1}*
- *ByteBuf dataReceived.* Rappresenta il buffer contente l'entry ricevuta, incluso il digest. Per la categorizzazione sono stati considerati gli attributi *ledgerID* ed *entryID*, relativi al contenuto del buffer in ricezione.
  - *{null, dataReceived.ledgerID != ledgerID || (dataReceived.ledgerID = ledgerID && dataReceived.entryID != entryID) || dataReceived.digestType != digestType, dataReceived.ledgerID = ledgerID && dataReceived.entryID == entryID && dataReceived.digestType == digestType}*
  - Analisy boundary-value: *{null, dataReceived.ledgerID != ledgerID || (dataReceived.ledgerID = ledgerID && dataReceived.entryID != entryID) || dataReceived.digestType != digestType, dataReceived.ledgerID = ledgerID && dataReceived.entryID == entryID && dataReceived.digestType == digestType}*

La *suite di test minimale* identificata è la seguente

- {HMAC, 0, 0, dataReceived.ledgerID = 0 & dataReceived.entryID = 1 & DataReceived.Type = HMAC}, {HMAC, -1, 1, dataReceived.ledgerID = 1 & dataReceived.entryID = 1 & DataReceived.Type = DUMMY}, {HMAC, 1, 1, dataReceived.ledgerID = 1 & dataReceived.entryID = 1 & DataReceived.Type = CRC32}, {HMAC, 1, 1, dataReceived.ledgerID = 1 & dataReceived.entryID = 1 & DataReceived.Type = HMAC},

### DigestManager – verifyDigestLac (ByteBuf dataReceived)
Il metodo permette al server, in ricezione, di verificare il digest dell'entry ricevuta dal client ed ottenere il campo *LAC* del pacchetto, rimuovendo in questo modo l'header dovuto al *digest*.

- *byte[] dataReceived.* {null, dataReceived.LAC < 0, dataReceived.LAC = 0, dataReceived.LAC = 1}
  - *Analisi boundary-value: {null, dataReceived.LAC = - 1, dataReceived.LAC = 0, dataReceived.LAC = 1}*

### DigestManager – verifyDigestAndReturnLastConfirmed (ByteBuf dataReceived)
Il metodo permette al server, in ricezione, di verificare il digest dell'entry ricevuta dal client ed ottenere il campo *lac* del pacchetto, rimuovendo in questo modo l'header dovuto al digest.

- *byte[] dataReceived.* Rappresenta il buffer contenente l'entry ricevuta.
  - *{null, dataReceived.entryID = entryID && dataReceived.ledgerID = ledgerID, dataReceived.ledgerID != ledgerID, dataReceived.digestType != digestType}*
  - Analisy boundary-value: *{null, dataReceived.entryID = entryID && dataReceived.ledgerID = ledgerID, dataReceived.ledgerID != ledgerID, dataReceived.digestType != digestType}*

### DigestManager – generateMasterKey (byte[] password)
Dato in input un'array di byte, corrispondente ad un array di una stringa, genera la master key necessaria per la generazione dei digest di tipo HMAC.

- *byte[] password. {null, string.len = 0, string.len > 0}*
  - Analisi boundary-value: *{null, string.len = 0, string.len = 1}*

## Category Partition - AVRO

### BinaryData – compare (byte[] b1, int s1, byte[] b2, int s2, Schema schema)
Il metodo compara due array di byte, utilizzando come riferimento per il confronto uno schema definito.

- *byte[] b1.* L'array di byte contenente dati generati secondo uno schema, indicato come *b1.Type.*
  - *{null, b1.schema = schema.Type, b1.schema != schema.Type}*
  - Analisi boundary-value: *{null, b1.schema = schema.Type, b1.schema != schema.Type}*
- *int s1.* Rappresenta l'offset all'interno di byte da cui iniziare a fare il confronto tra i due array.
  - *{<0, =0, >0}*
  - Analisi boundary-value: *{-1, 0, 1}*
- *byte[] b2.* Rappresenta l'array di byte contenente dati generati secondo uno schema, indicato come *b2.Type*. Il partizionamento è stato effettuato andando a considerare le possibili combinazioni dei parametri *Schema* e *Type* dei due arrat di byte.
  - *{null, b1.schema = b2.schema and schema=schema.Type, b1.schema = b2.schema and schema!=schema.Type || b1.schema != b2.schema}*
  - Analisi boundary-value: *{null, b1.schema = b2.schema and schema=schema.Type, b1.schema = b2.schema and schema!=schema.Type || b1.schema != b2.schema}*
- *int s2.* Rappresenta l'offset all'interno dell'array di byte da cui iniziare a fare il confronto. Il partizionamento è stato effettuato andando a considerare il valore di s2 rispetto all'attributo s1.
  - *{< s1, = s1, > s1}*
  - Analisi boundary-value: *{s1 - 1, s1, s1 + 1}*
- *Schema schema.* Rappresenta lo schema di riferimento per il confronto fra i due array di byte.
  - *{null, ARRAY, BOOLEAN, BYTES, DOUBLE, ENUM, FIXED, FLOAT, INT, LONG, MAP, NULL, RECORD, STRING, UNION}*
  - Analisi boundary-value: *{null, ARRAY, BOOLEAN, BYTES, DOUBLE, ENUM, FIXED, FLOAT, INT, LONG, MAP, NULL, RECORD, STRING, UNION}*

### SpecificData – getSchema (java.lang.reflect.Type type)
Il metodo restituisce lo *schema* associato alla tipologia di classe Java in input.

- *java.lang.reflect.Type type*

- *{null, Integer, Boolean, Void, Long, Float, Double, ByteBuffer, String, Collection, Map, OtherType}*
- Analisy boundary-value: *{null, Integer, Boolean, Void, Long, Float, Double, ByteBuffer, String, Collection, Map, OtherType}*

## Implementazione test

Per l'implementazione dei test è stato utilizzato il framework **JUnit** con il runner **Parameterized**. Si tratta di un runner standard che implementa i test parametrizzati, andando a definire una matrice *mxn*, dove *m* rappresenta il numero di test da eseguire ed *n* il numero di parametri necessari per ogni test. Sono stato associate ad alcuni metodi le annotazioni fornite da JUnit **@Before** ed **@After** che, rispettivamente, permettono di eseguire delle operazioni prima e dopo ogni test.

Come indicato per alcuni metodi, la problematica principale riscontrata nell'implementazione dei casi di test è stata la non conformità tra quanto previsto dalle specifiche, ottenute dalla documentazione presente, e l'effettiva implementazione dei metodi. Le motivazioni sono differenti per i diversi tipi di progetti. Questo porta, a differenza di quanto si era preventivato con l'analisi basata su *category partition* e dei *boundary values*, ad avere una suite di test che, una volta implementata, non risulta minimale.

### TestBinaryDataCompare - AVRO

Nella stesura della *category partition,* considerati i parametri di input del metodo, era stata fatta l'assunzione che la comparazione dei due array avvenisse in riferimento allo *schema* indicato; dunque se i due array di byte fossero stati uguali, ma diversi dallo schema di riferimento, allora la comparazione non sarebbe dovuta andare a buon fine. Nell'implementazione invece, se i due array di byte dati in input sono stati generati secondo lo stesso schema e contengono gli stessi valori, allora la comparazione va a buon fine a prescindere dal parametro *schema*. Per l'esecuzione del test sono stati aggiunti i metodi:

- *getS(Schema.Type type):* il metodo restituisce l'istanza dell'oggetto *schema*, necessario alla funzione *compare().* La creazione dell'oggetto schema avviene in due modi differenti.
  - Per i tipi di dato semplici, viene utilizzato il metodo *schema.create()*
  - Per i tipi di dato complesso invece, viene utilizzato il metodo *parse()* che, data una stringa in input, genera un determinato schema. L'individuazione delle stringhe da dare in input è stata effettuata partendo dalla documentazione del progetto.
- *getBS(Schema.Type type, boolean createB1):* il metodo restituisce un array di byte generato secondo lo schema dato in input. Per permettere di ottenere degli array differenti viene utilizzato il parametro booleano, tramite il quale vengono distinti i casi in cui si utilizzano dei valori piuttosto che altri.

Inoltre è stata aggiunta la classe ***BinaryDataUtils,*** che offre dei metodi di supporto per la creazione degli array di byte relativi ad un determinato tipo di schema.

### TestSpecificDataGetSchema - AVRO

Per l'esecuzione del test è stato implementato il metodo

- *getSchemaType(String classTypeString):* il metodo restituisce il tipo di dato Java, necessario come parametro per il metodo *getSchema().*
  - Per i tipi di dato semplice viene restituito, come tipo di dato, la classe Java dell'oggetto (*tipoOggetto.class*)
  - Per i tipi di dato complessi invece, *map* e *collection*, viene utilizzata la reflection su degli attributi privati della classe di test per ottenere la tipologia delle due variabili a runtime.

### TestDigestManager* - Bookkeeper

Nella stesura della *category partition,* considerati i parametri di input del metodo, si era stata fatta l'assunzione per cui il valore di *entryID* non potesse essere negativo, così come il valore di *ledgerID* e la condizione *LAC > entryID* tra i due parametri. Nell'implementazione, però, non è presente nessun controllo sul valore assunto dai tre parametri. Questo può essere dovuto alla posizione della classe nello stack di comunicazione, trovandosi ad un livello molto basso che precede l'effettivo invio sulla rete. Dunque è facile immaginare come tutti i controlli evidenziati in precedenza siano presenti ad un livello più alto. Per l'esecuzione dei test sono stati implementati diversi metodi per la generazione delle entry necessarie al metodo assert dei test implementati.

- *generateEntry(int length):* il metodo restituisce un *ByteBuf* relativo alla singola entry. Per la generazione, vengono scritti in sequenza le informazioni della singola entry come indicato dalla documentazione.

- *generateBadEntry(int length):* il metodo restituisce un *ByteBuf* relativo alla singola entry. A differenza del metodo precedente, viene omessa la scrittura dell'header contente le informazioni della entry, andando a scrivere solamente il buffer di dati.
- *generateLacWithDigest(int lacID, DigestType digestType, long ledgerID, boolean useV2Protocol):* il metodo genera un *Lac*, utilizzando il metodo offerto dalla classe *Digest*
- *generateBadLacWithDigest(int lacID):* il metodo genera un *Lac* in modo malformato.
- *generateLastAddConfirmed(int lacID, DigestType digestType, long ledgerID, boolean useV2Protocol):* il metodo restituisce una *ByteBufList* contente il LAC del pacchetto e l'array di byte relativi ai dati contenuti nella entry.

### TestBufferedChannel* - Bookkeeper
Per l'esecuzione dei test sono stati implementati dversi metodi per la generazione delle entry necessarie.
- *generateEntryWithWrite():* il metodo restituisce una *entry*, con il buffer di dati popolato da scritture.
- *generateEntryWithoutWrite():* il metodo restituisce una *entry*, con il buffer di dati non popolato da scritture.
- *generateEntryWithWriteResetIndex(int length):* il metodo restituisce una entry andando ad effettuare il reset del writer index prima della scrittura dei dati. E' stata aggiunta questa chiamata al metodo dell'oggetto per la copertura della linea 249.

## Adeguatezza test

Per la misurazione delle metriche relative all'adeguatezza dei test è stato utilizzato il plugin **JaCoCo**. Per l'inclusione all'interno dei due progetti è stato modificato il file *pom.xml* principale, in tutti e due i casi, ed è stato creato il modulo *jacoco-test-report-aggregator* che si occupa della generazione di un report aggregato riguardante la copertura dei test implementati. Per l'integrazione con *SonarCloud*, al modulo citato ed ai moduli contenenti i test da eseguire, è stato aggiunto alle *properties* già presenti, quella relativa all'integrazione son Sonar.

Le metriche di adeguatezza per i test scelte sono **line coverage** e **condition coverage.** Non avendo testato tutti i metodi della classe in alcuni casi, questo potrebbe portare a dei risultati falsati; il valore relativo all'incremento della singola classe non rispecchia quello dei metodi testati.

### BinaryData - AVRO
Le misurazioni effettuate sulla classe *BinaryData*, prima e dopo l'introduzione di ulteriori casi di test, con l'esecuzione del test *TestBinaryDataCompare* sono disponibili in tabella 1. Per l'incremento dei due valori sono stati introdotti due casi di test. E' stato aggiunto un caso di test per la comparazione di due array di byte entrambi generati in modo coerente con lo schema di riferimento, ma con offset differente; l'altro caso di test vede la comparazione di due bity di array, uguali tra di loro, generati in modo coerente con lo schema di riferimento (*RECORD*).

### SpecificData - AVRO
Le misurazioni effettuate sulla classe *SpecificData*, prima e dopo l'introduzione di ulteriori casi di test, con l'esecuzione del test *TestSpecificDataGetSchema* sono disponibili in tabella 2. Sono stati aggiunti dei casi di test per incrementare la *branch coverage* del metodo, in quanto con la suite minimale non venivano considerati gli OR presenti da linea 354 a linea 364. E' stato inoltre aggiunto un caso di test per la copertura della linea 378, andando a passare come input un Array però non di stringhe ma di interi, in modo da entrare nella condizione e sollevare l'eccezione.

### DigestManager - Bookkeeper
Le misurazioni effettuate sulla classe *DigestManager*, prima e dopo l'introduzione di ulteriori casi di test, con l'esecuzione dei test *TestDigestManager** sono disponibili in tabella 3. Per l'incremento della condition coverage è stato modificata l'implementazione dei casi di test per andare a coprire la condizione a linea 154. Sono stati inoltre aggiunti dei casi di test per la copertura delle condizioni di controllo tra l'uguaglianza di *digestID* e *ledgerID*.

### BufferedChannel - Bookkeeper
Le misurazioni effettuate sulla classe *BufferedChannel*, prima e dopo l'introduzione di ulteriori casi di test, con l'esecuzione dei test *TestBufferedChannel** sono disponibili in tabella 4. La suite minimale non è assolutamente adatta alla complessità della classe, come è possibile vedere dal valore della *condition coverage*, sopratutto. Sono stati introdotti sei test per la coverage per il metodo *read* ed un caso di test per il

metodo *write*. In tutti e sette i casi l'obiettivo è stato quello di andare a stimolare il sistema in situazioni particolari, non rilevale dalla category partition e successiva analisi dei boundary-values.

## Mutation testing

Per la misurazione dell'adeguatezza dei test rispetto alle mutazioni è stato utilizzato il plugin **PIT Mutation Testing**. Per la sua inclusione all'interno dei due progetti è stato modificato il file *pom.xml* dei sottomoduli di cui si vogliono analizzare i test. Nella cartella *Mutation report*, sono disponibili i report navigabili in formato *html* di riferimento alle linee di codice di seguito illustrate.

Come già visto per la coverage del codice, il numero di mutazioni e la percentuale di mutanti uccisi sono relativi all'intera classe; questo falsa l'incremento della robustezza dell'insieme di test nei casi in cui questi ultimi sono riferiti solamente ad alcuni metodi e non a tutta la classe.

### SpecificData - AVRO

L'esecuzione di PIT ha portato ai risultati disponibili in tabella 6. Non sono stati introdotti casi di test per il rilevamento di ulteriori mutazioni, in quanto tutte quelle coperte vengono *uccise.*

### BinaryData - AVRO

L'esecuzione di PIT ha portato ai risultati disponibili in tabella 5. Con l'introduzione dei nuovi casi di test è stato possibile individuare ulteriori 16 mutazioni, portando il totale a 116 su 219. Qui di seguito è presente il riferimento alle mutazioni, coperte dall'esecuzione dei test, che non sono state uccise.

- *Linea 76:* equivalente al SUT rispetto a strong mutation, ma non rispetto a weak mutation, in quanto rimuovendo la chiamata al metodo *clear()* lo stato interno dei decoder *d1* e *d2* differisce rispetto all'esecuzione senza mutazioni.
- *Linea 142-143:* equivalente al SUT rispetto a strong mutation, ma non rispetto a weak mutation, in quanto gli execution path percorsi sono differenti.
- *Lina 158-159:* equivalente al SUT rispetto a strong mutation, ma non rispetto a weak mutation in quanto si ha una differenza dello stato interno dei decoder *d1* e *d2* rispetto all'esecuzione senza mutazioni.
- *Linea 185:* equivalente al SUT rispetto a strong mutation, ma non rispetto a weak mutation in quanto viene modificato lo stato interno delle variabili *i* e *j* e viene eseguito un path di esecuzione differente.
- *Linea 321-330:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto vengono modificati alcuni valori dell'array di byte buf[], ma il metodo restituisce il valore pos-start.
- *Linea 352-376:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto viene modificato il valore della variabile buf[].
- *Linea 388:* equivalente al SUT rispetto a strong e weak mutation
- *Linea 399-401:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto vengono modificati i valori dell'array di byte buf[], ma il metodo restituisce sempre il valore 4.
- *Linea 424-426:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto vengono modificati i valori dell'array di byte buf[], ma il metodo restituisce sempre il valore 8.

### DigestManager - Bookkeeper

L'esecuzione di PIT ha portato ai risultati disponibili in tabella 7. Con l'introduzione dei nuovi casi di test è stato possibile individuare 3 ulteriori mutazioni, portando il totale a 42 su 48. Qui di seguito è presente il riferimento alle mutazioni, coperte dall'esecuzione dei test, che non sono state uccise.

- *Linea 87:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto cambia il path di esecuzione. Il risultato del test è equivalente in quanto l'assert viene fatto sul contenuto dell'array, che risulta essere vuoto, anche se l'istanza dell'oggetto restituito è effettivamente differente.
- *Linea 102-105:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto la mutazione modifica la tipologia di buffer allocato, e questo non incide sull'esito del test.
- *Linea 128:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto cambia il path di esecuzione. Il risultato del test è equivalente; il valore booleano cambia solamente il tipo di buffer che viene creato.

### BufferedChannel - Bookkeeper

L'esecuzione di PIT ha portato ai risultati disponibili in tabella 8. Per l'individuazione di ulteriori mutazioni sono stati aggiunti casi di test, indicati nei file .java della classe di test sotto al commento *Mutation*. Con l'introduzione dei nuovi casi di test è stato possibile individuare 4 ulteriori mutazioni, portando il totale a 37 su 60. Qui di seguito è presente il riferimento alle mutazioni, coperte dall'esecuzione dei test, che non sono state *uccise*.

- *Linea 94-98:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto cambia il path di esecuzione. Il risultato del test è equivalente, trattandosi di un metodo *void*.
- *Linea 127:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation, in quanto viene modificato il valore della variabile *position*.
- *Linea 136:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation. Stesso risultato osservabile ma path di esecuzione differente.
- *Linea 225-232:* equivalente al SUT rispetto a strong mutation ma non rispetto a weak mutation.
- *Linea 251:* equivalente al SUT rispetto a weak e strong mutation. Stesso risultato osservabile e stesso path di esecuzione, in quanto la variabile *writeBuffer* non è mai uguale a **null** nei test considerati.
- *Linea 257:* equivalente al SUT rispetto a weak e strong mutation. Stesso risultato osservabile e stesso path di esecuzione, in quanto la variabile *positionInBuffer* è sempre uguale a 0 nei test considerati.
- *Linea 267:* equivalente al SUT rispetto a weak e strong mutation. Stesso risultato osservabile e stesso path di esecuzione, in quanto il numero di byte letti è sempre maggiore di 0 nei test considerati.

*Tabella 1: Line e condition coverage - BinaryData - AVRO*

| | *Line* coverage | Condition coverage |
|---|---|---|
| Suite minimale | 62% | 40.8% |
| Aggiunta test coverage | 73.6% | 55.3% |

*Tabella 2: Line e condition coverage - SpecificData - AVRO*

| | *Line* coverage | Condition coverage |
|---|---|---|
| Suite minimale | 33% | 35% |
| Aggiunta test coverage | 37.5.% | 39.9% |

*Tabella 3: Line e condition coverage - DigestManager - Bookkeeper*

| | *Line* coverage | Condition coverage |
|---|---|---|
| Suite minimale | 86% | 85.2% |
| Aggiunta test coverage | 92.5% | 92.6% |

*Tabella 4: Line e condition coverage - BufferedChannel - Bookkeeper*

| | *Line* coverage | Condition coverage |
|---|---|---|
| Suite minimale | 58.3% | 37.5% |
| Aggiunta test coverage | 80.9% | 71.1% |

*Tabella 5: Mutation coverage - BinaryData - AVRO*

| | Mutation coverage |
|---|---|
| Suite minimale con coverage | 100/219 (46%) |
| Aggiunta test mutazioni | 116/219 (53%) |

*Tabella 6: Mutation coverage - SpecificData - AVRO*

| | Mutation coverage |
|---|---|
| Suite minimale con coverage | 39/142 (27%) |
| Aggiunta test mutazioni | |

*Tabella 7:  Mutation coverage - DigestManager - Bookkeeper*

| | Mutation coverage |
|---|---|
| Suite minimale con coverage | 39/48 (81%) |
| Aggiunta test mutazioni | 42/48 (88%) |

*Tabella 8: Mutation coverage - BufferedChannel - Bookkeeper*

| | Mutation coverage |
|---|---|
| Suite minimale con coverage | 33/60 (56%) |
| Aggiunta test mutazioni | 37/60 (62%) |