INTRODUZIONE

Lo scopo di questo lavoro è quello di implementare varie tecniche di testing e di adeguatezza dei test su due progetti open source che sono Apache-Bookkeeper e Apache-Tajo. In generale gli step realizzati per entrambi i progetti è lo stesso, e sono stati eseguiti su OS Ubuntu 20.04:

1. Installazione software maven e open-jdk necessari per la build del progetto.
2. Eseguito il clone della repository github dei progetti in locale.
3. Eliminazione di tutti i test già implementati e build del codice in locale.
4. Inserito file travis.yml configurato in modo opportuno per effettuare l'integrazione con Travis-CI e SonarCloud.
5. Scelta di 2 classi per ogni progetto e realizzazione dei primi test utilizzando la tecnica di Category Partition per l'individuazione della test suite minima utilizzando i framework JUnit4 e Mockito per la creazione di mock ove necessario.
6. Integrazione framework jacoco (strumento per verifica di adeguatezza basato su control-flow graph) e incremento della test suite minima per migliorare i vari criteri di coverage, con l'aiuto anche della visualizzazione dei risultati su SonarCloud.
7. Integrazione di pitest per eseguire mutation testing e migliorare ancora l'adeguatezza della test suite uccidendo quante più mutazioni possibili.

In generale in tutti i test le parti commentate con la dicitura "*FOR COVERAGE*" indicano che quella parte di test è stata sviluppata a seguito dell'analisi del report di jacoco per aumentare la copertura di metriche di adeguatezza, mentre la dicitura "*MUTATION N° KILLED*" indica invece parti implementate per uccidere una mutazione generata dall'analisi con pitest. In generale i test sono eseguiti quando si esegue la build di uno dei due sistemi, al contrario invece pitest non viene eseguito ad ogni build. A seguito della build di un progetto tramite il comando "*mvn install*" (necessario per far trovare a pitest le classi da mutare), per generare le cartelle contenenti i report di si deve eseguire il comando "*mvn org.pitest:pitest-maven:mutationCoverage*".

Andiamo ora a vedere nel dettaglio, per ognuno dei due progetti, come sono stati realizzati tutti questi passaggi.

APACHE BOOKKEEPER

BUILD LOCALE E CONFIGURAZIONE TRAVIS E SONARCLOUD

Una volta effettuato il clone della repository github in locale e aver cancellato tutti i test presenti, il sistema non ha avuto bisogno di molte altre configurazioni. Semplicemente impostando la open-jdk11 e lanciando il comando *"mvn clean package"* il processo di build viene completato con successo. Il file *".travis.yml"* viene aggiunto per integrare ogni commit su github con Travis-CI e Soracloud. Nel file mostrato in Fig.1 possiamo analizzare cosa chiediamo di eseguire e come configuriamo la macchina virtuale che eseguirà la build su Travis:

1. Linguaggio da utilizzare Java e permessi di superuser non necessari.
2. Il campo addons necessario per inserire i dati per l'integrazione con Sonarcloud.
3. Lo script da eseguire per la build. Possiamo notare come viene eseguita la fase di package, e non di install, in quanto è quella sufficiente per far realizzare i file necessari a jacoco per la visualizzazione della coverage in SonarCloud. In aggiunta ci sono altri comandi sempre per l'integrazione con quest'ultimo.

I framework Junit e Mockito erano già integrati all'interno del progetto quindi non è stato necessario aggiungere nulla per integrarli. Diverso invece è stato per Jacoco (Fig.2) è stato creato anche un modulo fittizio per aggregare i risultati ottenuti dai vari moduli, e per Pitest (Fig.3) dove si è dovuto specificare anche le classi su cui eseguire le mutazioni (per evitare di mutare anche tutte le classi non soggette a test) e i test da eseguire.

CATEGORY PARTITION E PRIMI TEST PER BLOCKINGMPSCQUEUE

La prima classe analizzata è *"BlockingMpscQueue<T>"* che da come si può leggere dalla documentazione (A Multi-Producer-Single-Consumer queue based on a org.jctools.queues.ConcurrentCircularArrayQueue) è basata su "*ConcurrentCircularArrayQueue".* Andando a vedere il costruttore in Fig.4 una cosa fondamentale che possiamo notare per l'analisi è che la capacità di questa viene sempre arrotondata alla più vicina potenza di 2 superiore rispetto all'intero passato nel costruttore. Nel test è presente un metodo con annotazione @Before che viene eseguito prima di ogni test e serve ad inizializzare una lista vuota da utilizzare poi all'interno del test. I primi due metodi testati sono il metodo *offer()* e il metodo *poll()* nel caso di test *"testOfferAndPoll()".*

```
public boolean offer(T e, long timeout, TimeUnit unit) throws InterruptedException
```

Questo metodo prova ad inserire un elemento nella coda finché non scade il timeout .Prende in input:

1. *T e:* in questo caso un oggetto di tipo Integer, ed è l'elemento da provare ad aggiungere alla lista.
2. *long timeout:* il tempo massimo da attendere nel tentativo di aggiungere l'elemento alla lista.
3. *TimeUnit unit:* un'enumerazione che indica l'unità di misura del timeout.

Per scegliere le classi di equivalenza ho applicato la tecnica di category partition nel seguente modo:

1. *int toAdd*: { < 0 ; 0 ; > 0 }. Essendo un intero ho semplicemente diviso in negativi, positivi e zero
2. *long timeout*: {< 0 ; 0 ; > 0}. Ho diviso il long in questo modo per avere un timeout valido, uno nullo, e uno invalido
4. *TimeUnit unit:* {SECONDS;NANOSECONDS;MINUTES;MILLISECONDS;MICROSECONDS;HOURS;DAYS}. Ho semplicemente scelto tutta l'enumerazione. (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html)

I seguenti casi di test derivati da questa partizione sono stati generati seguendo il metodo unidimensionale (deve esserci un rappresentante per ogni classe di equivalenza):

```
"1,100,MILLISECONDS", // toAdd > 0 ; timeout > 0 ; unit = MILLISECONDS
"0,1,SECONDS", // toAdd = 0 ; timeout > 0 ; unit = SECONDS
"-1,100,NANOSECONDS", // toAdd < 0 ; timeout > 0 ; unit = NANOSECONDS
"1,0,MINUTES", // toAdd > 0 ; timeout = 0 ; unit = MINUTES
"1,100,MICROSECONDS", // toAdd > 0 ; timeout > 0 ; unit = MICTOSECONDS
"0,-1,HOURS",// toAdd = 0 ; timeout < 0 ; unit = HOURS
"0,-1,DAYS" // toAdd = 0 ; timeout < 0 ; unit = DAYS
```

Ad ogni inserimento viene controllato il valore di ritorno che essendo un booleano può essere true o false, il primo se l'elemento è stato aggiunto alla coda e il secondo se l'inserimento non è andato a buon fine. Ognuno dei precedenti test viene eseguito fino a riempire la coda ed ogni volta viene restituito true, dopodichè si prova un altro inserimento e ci si aspetta che venga restituito false. All'interno dello stesso metodo viene testata anche la funzionalità del metodo poll.

```
public T poll(long timeout, TimeUnit unit) throws InterruptedException
```

Come input prende gli stessi valori passati a *offer()*, non serve quindi un ulteriore category partition, e restituisce il valore estratto dal primo elemento della coda o null altrimenti. Il test viene effettuato estraendo uno ad uno gli elementi dalla coda fino a svuotarla, per poi effettuare un ulteriore chiamata ed attendere lo scadere del timeout e la restituzione di null.

Passiamo poi al testing dei metodi *take()* e *put()* con il caso di test *"testTakeAndPut()".*

```
public void put(T e) throws InterruptedException
```

```
public T take() throws InterruptedException
```

Siamo in un caso semplice e anche simile a *testOfferAndPoll().*Infatti non troviamo nè timeout e nemmeno una valore di ritorno, di conseguenza bisogna testare la lunghezza della coda per verificarne il corretto

funzionamento. Semplicemente si prova ad eseguire una *put()* più volte e verificare che la lunghezza finale coincida con gli elementi aggiunti (grazie al metodo *size()* che restituisce la lunghezza attuale della coda). Dopodiché si esegue una serie di *take()* fino a svuotare la coda e si verifica che ciò sia avvenuto.

Passiamo poi a testare il metodo *drainTo()* che serve a trasferire dalla coda gli elementi in una lista. Se la lista non è abbastanza grande arriva a riempirla e i restanti valori vengono lasciati nella coda. Il valore di ritorno è la dimensione della lista al termine del metodo meno la sua dimensione iniziale. Di questo metodo ne esistono due varianti: uno con un limite di elementi da trasferire, ed uno senza che tenta di trasferire tutta la coda.

```
public int drainTo(Collection<? super T> c, int maxElements)
```

In *testDrainTo()* viene effettuato il test del metodo senza limite di elementi e la category partition viene fatta semplicemente nel seguente modo :

1. Collection<T> c: {lista vuota, lista riempita a metà, null}. Lista all'interno della quale trasferire gli elementi

In *testDrainToLimited()* passiamo a testare la versione con il limite. La category partition è la seguente:

1. Collection c : {lista vuota, lista riempita a metà, null}.
2. int maxElements : { < 0 ; 0 ;0 < e <= realSize ; > realSize} limite degli elementi da trasferire alla lista dove realSize è la dimensione della coda.

In questo caso viene utilizzato un approccio multidimensionale, ovvero ogni classe di maxElements viene testata con ogni classe di Collection. In questo modo la test suite cresce molto di dimensione ma abbiamo una maggiore densità dei test.

Ultimo test sviluppato è *testRemainingCapacity()* che serve a testare il metodo *remainingCapacity()* che semplicemente restituisce la capacità rimanente della coda. Il test viene effettuato con una coda vuota, riempita a metà, piena e una null.

AMPLIAMENTO TEST SUITE E ADEGUATEZZA BLOCKINGMPSCQUEUE

A seguito della scrittura di questa test suite minimale, vedendo i risultati di coverage su SonarCloud (linee coperte e condizioni coperte) si è cercato di ampliare il numero di test in maniera da coprire ciò che era rimasto scoperto. Partendo da *offer()* e *poll()* ho introdotto un ulteriore test chiamato *testInterrupt()* per testare il comportamento in caso il thread venga segnato come interrotto durante l'esecuzione.Nei due metodi infatti è presente un controllo su una flag che determina se un thread è interrotto o meno, e questa eventualità non era stata coperta dalla test suite sviluppata precedentemente. Grazie a questa aggiunta si possono vedere (link sonar : BlockingMpscTest SonarCloud) i risultati mostrati da SonarCloud dai quali calcoliamo le seguenti metriche di adeguatezza. Metodi *offer()* e *poll()* :

1. Statement Coverage = 1 : Tutte le linee di codice sono state coperte in entrambi.
2. Branch Decision Coverage = 1 : Tutti i branch sono stati esplorati in entrambi.
3. Condition Coverage = 1 : Tutte le condizioni coperte da almeno un caso true e uno false
4. Branch Condition Coverage = 1
5. Multiple Condition Coverage = 1 : anche se non ha senso essendo tutte condizioni semplici

Passando ad i metodi *put()* e *take()* anche per questi è stato introdotto il test chiamato *testTakeAndPutInterrupt()* per aumentare la coverage. La funzionalità e lo scopo è analogo a *testInterrupt()*. Sono stati inoltre aggiunti altri due test che sono *testTakeTimeout()* e *testPutTimeout()*. Questi due test sono dotati di un timeout che scadrà sempre e farà fallire l'esecuzione di tali test non raggiungendo mai l'asserzione di fail presente. Questo è stato fatto in quanto quando si prova ad eseguire le operazioni *put()* o *take()* su una coda che attualmente non le supporta, queste rimangono bloccate in un loop infinito che può essere interrotto solo attraverso un timeout. Il loro inserimento è comunque stato importante in quanto ha permesso di portare tutte le metriche elencate sopra ad 1 anche per questi due metodi. L'unico inconveniente (e succederà anche per altri test nel seguito dell'analisi) è che lo scadere del timeout porta il test a fallire. E se è vero che questo può essere aggirato e la build portata a conclusione semplicemente

aggiungendo delle configurazioni nel pom, la stessa cosa non si può dire quando in seguito effettueremo mutation testing con pitest. Questo framework, infatti, necessita che nessuno dei test fallisca per poter funzionare, ed è quindi necessario disabilitare questi test prima di farlo eseguire (commentandoli o inserendo prima di ciascuno l'annotazione *@Ignore*) altrimenti la sua analisi non andrà a buon fine.

Per quanto riguarda invece i metodi più semplici come *drainTo()* e *remainingCapacity()* l'analisi non ha prodotto risultati d'interesse come ci si poteva aspettare, essendo metodi senza diversi branch e con un solo blocco. L'unica metrica di adeguatezza di cui ha senso parlare in questo caso è la statement coverage che è pari ad 1.

MUTATION TESTING BLOCKINGMPSCQUEUE

A questo punto per incrementare ulteriormente l'adeguatezza della test suite è stato utilizzato il framework pitest, con le sue mutazioni predefinite, per fare mutation testing sulla classe in questione. I risultati sono visibili riassunti in Fig.5 e il file completo è disponibile nella cartella pit-reports-book-common presente nel repository github(come anche per le altre classi). Le mutazioni riguardanti i metodi testati sono solo quelle racchiuse nel riquadro. La **mutazione 104** prevede il ritorno di un valore null invece del valore preso dalla *take()* dalla coda, ed era sopravvissuta in quanto non era stato controllato il valore di ritorno ma solo la lunghezza della coda finale. E' bastato controllare che il valore restituito fosse uguale a quelli inseriti e la mutazione è stata uccisa. Le **mutazioni 53 e 96** invece cambiano la condizione di scadenza del timeout da maggiore a maggiore o uguale nei metodi *offer()* e *poll().* Questo in realtà non porta nessun cambiamento all'interno dell'esecuzione del metodo e del risultato ottenuto in quanto una volta superata la soglia prestabilita del timout il metodo termina correttamente(al massimo compiendo un ciclo di esecuzione in più nel caso in cui il tempo trascorso sia esattamente uguale al timeout impostato). Siamo quindi di fronte a due mutazioni che comportano una weak mutation (non cambia né il flusso d'esecuzione né l'output) che sono equivalenti al SUT. Le ultime due di cui parlare sono le **mutazioni 40 e 71** dove viene negata la condizione in cui si verifica se il Thread è stato interrotto nei metodi *take()* e *put().* In questo caso i test fatti partire da pitest vengono interrotti a causa del rilevamento di un loop infinito, e purtroppo quindi il caso di test non viene completato. A causa di ciò, e anche del fatto che i timeout sui metodi sono rimossi durante l'esecuzione di pitest, le due mutazioni purtroppo non vengono uccise. Infine tutte le altre mutazioni erano risultate già uccise con la test suite precedente. Partendo da questi dati quindi possiamo calcolare il MutationCoverage della nostra test suite che è pari a $\frac{20}{24-2} = 0.9$

CATEGORY PARTITION E PRIMI TEST PER BUFFEREDCHANNEL

La prossima classe sottoposta a test è BufferedChannel. Un buffered channel è essenzialmente un buffer collegato ad un file che ad ogni scrittura scrive sul buffer locale e, se i byte raggiungono un limite prestabilito o la capienza massima, allora vengono scritti sul file. I metodi analizzati di questa classe sono quelli fondamentali, ovvero *write()* e *read().* Per prima cosa analizziamo il costruttore per capire com' è fatto questo oggetto e cosa significano i vari parametri che contiene.

```
public BufferedChannel(ByteBufAllocator allocator, FileChannel fc, int writeCapacity,
int readCapacity, long unpersistedBytesBound) throws IOException
```

Vediamo in ordine che come input prende: un buffer su cui scrivere e leggere, un FileChannel collegato ad un file sul quale fare il flush del buffer in determinate condizioni, la capacità del buffer in scrittura e lettura (in genere uguali negli esempi sviluppati) e un long finale che sta ad indicare il limite massimo di byte che possono essere presenti nel buffer in maniera non persistente. Dalla documentazione di quest' ultimo attributo presente nel codice stesso, possiamo intuire anche che questo valore può essere zero, e che quindi possiamo anche non impostare questo limite.

```
* if unpersistedBytesBound is non-zero value, then after writing to
* writeBuffer, it will check if the unpersistedBytes is greater than
* unpersistedBytesBound and then calls flush method if it is greater.
```

Partiamo ad analizzare la category partition effettuata per *testWrite()*, funzione di test per il metodo *write()* :

1. capacity: {<= 0 ; > 0} : capacità del BufferedChannel

2. unpersistedBytesBound : {<= 0 ;0 < x <= capacity ; > capacity} : numero di byte non persistenti max
3. lenght : {<= 0 ; = capacity && = unpersistedBytesBound ; < unpersistedBytesBound && < capacity ; unpersistedBytesBound < x < capacity ; capacity < x < unpersistedBytesBound ; > capacity && > unpersistedBytesBound} : lunghezza del buffer di input da scrivere nel BufferedChannel

Da questa analisi sono stati sviluppati i seguenti casi di test minimi secondo il metodo unidimensionale:

"40,40,40", // capacity > 0 ; Bound <= capacity : length = capacity && length = unpersistedBytesBound
"40,40,30", // capacity > 0 ; Bound <= capacity : length < capacity && length < unpersistedBytesBound
"40,41,30", // capacity > 0 ; Bound > capacity : length < capacity && length < unpersistedBytesBound
"40,30,35", // capacity > 0 ; Bound <= capacity ; Bound < length < capacity
"40,50,45", // capacity > 0 ; Bound > capacity ; capacity < length < Bound
"40,10,51", // capacity > 0 ; Bound <= capacity ; length > capacity && length > Bound
"40,0,10", // capacity > 0 ; Bound <= 0 ; Bound < length < capacity
"0,0,1", // capacity <= 0 ; Bound <= 0 ; Bound < length < capacity ----> BUG con capacità nulla
"40,10,0" // capacity > 0 ; Bound <= capacity ; length <= 0

Ad ogni scrittura poi vengono controllati il numero di bytes presenti ancora nel buffer e quelli invece che sono stati scritti nel file channel. Una particolarità è stata trovata nel caso di test in cui la capacità del BufferedChannel è nulla. Dalla documentazione del metodo *write()* infatti si può notare le seguente frase : `@throws IOException if a write operation fails.` Nel caso di test appena menzionato invece la funzione non ritorna alcun risultato e non lancia alcuna eccezione, ma continua in un loop infinito non essendoci nessun controllo sulla capacità del BufferedChannel.

Analizziamo ora la category partition di ***testRead().*** funzione di test per il metodo *read()* :

1. int maxLen : {> 0; <= 0} : lunghezza del BufferedChannel da cui leggere
2. int pos : {<0 ; 0 ; 0 < x <= maxLen ; > maxLen} : posizione da cui iniziare a leggere
3. int length : {<0 ; 0 ; 0 < x <= |maxLen-pos| ; > |maxLen-pos|} : quanti byte si vogliono leggere. E' necessario il modulo in quando la posizione potrebbe essere oltre la maxLen e quindi quel valore uscire negativo.

Da questa analisi sono stati ricavati i seguenti casi di test seguendo il metodo unidimensionale.

"40,0,20", //maxLen > 0; pos = 0 ; 0 < length <= |maxLen-pos|
"40,30,20", //maxLen > 0; 0 < pos <= maxLen ; length > |maxLen-pos|
"40,10,0", // maxLen > 0 ; 0 < pos <= maxLen ; length = 0
"20,30,10", // maxLen > 0 ; pos > maxLen ; 0 < length <= |maxLen-pos|
"40,-1,10", // maxLen > 0 ; pos < 0 ; 0 < length <= |maxLen-pos|
"40,20,30", // maxLen > 0 ; 0 < pos <= maxLen ; length > |maxLen-pos|
"40,20,-1", // maxLen > 0 ; 0 < pos <= maxLen ; length < 0

Nella test suite minimale manca il caso maxLen <= 0, questo perché comporterebbe una scrittura (la quale precede l'operazione di read) su un BufferedChannel con lunghezza nulla e quindi un loop infinito già testato nel test precedente della write().

AMPLIAMENTO TEST SUITE E ADEGUATEZZA BUFFEREDCHANNEL

A seguito della realizzazione della test suite minimale, grazie all'analisi fornita da SonarCloud (link BufferedChannelTest SonarCloud) la test suite è stata ampliata per aumentare la coverage. Per ***testWrite()*** non sono stati aggiunti ulteriori casi di test in quanto *write()* era già sufficientemente coperto, mente in ***testRead()*** sono stati aggiunti i seguenti casi di test per incrementare l'adeguatezza :

"40,50,1", // maxLen > 0 ; pos > maxLen ; 0 < length <= |maxLen-pos| FOR COVERAGE
"50,0,20", // maxLen > 0 ; 0 < pos <= maxLen; 0 < length <= |maxLen-pos| FOR COVERAGE
"30,0,200", // maxLen > 0 ; pos > maxLen ; length > |maxLen-pos| FOR COVERAGE

Queste aggiunte sono servite principalmente a coprire alcune condizioni non ancora verificate da nessun test (o almeno tutte quelle raggiungibili). Un altro test aggiunto per aumentare la coverage e coprire una condizione nella *read()* è ***testNullWriteBuffer().*** In questo test si prova ad effettuare una lettura su un BufferedChannel che ha il suo writeBuffer interno inizializzato a null. Siccome questo è un parametro

configurato direttamente all'interno del costruttore e non scelto dall'utente, si è dovuto ricorrere all'utilizzo del framework Mockito per generare uno stub per questo caso di test che restituisca il valore null ogni volta che nel costruttore si cerca di inizializzare writeBuffer.

Le metriche di adeguatezza del metodo *write()* sono le seguenti :

1. Statement Coverage = 1 : Tutte le linee di codice sono state coperte.
2. Branch Decision Coverage = 1 : Tutti i branch sono stati esplorati.
3. Condition Coverage = $\frac{9}{10} = 0.9$: Una condizione non è coperta nel caso false
4. Branch Condition Coverage = $\frac{9+5}{10+5} = 0.933$: Manca la condizione del caso false precedente
5. Multiple Condition Coverage = 0.9 : come la condition coverage essendo solo condizioni semplici

Le metriche di adeguatezza del metodo *read()* invece sono :

1. Statement Coverage = $\frac{35}{36-1} = 1$: Tutte le linee di codice sono state coperte tranne una non raggiungibile
2. Branch Decision Coverage = $\frac{6}{7-1} = 1$: Tutti i branch sono stati esplorati tranne uno non raggiungibile
3. Condition Coverage = $\frac{11}{12-1} = 1$: Una condizione semplice non è soddisfacibile nel caso true
4. Branch Condition Coverage = $\frac{11+6}{(12-1)+(7-1)} = 1$: Manca la condizione del caso true precedente
5. Multiple Condition Coverage = $\frac{16}{18-1} = 0.94$: Una condizione semplice non è raggiungibile nel caso true

Come si può notare anche dalle metriche appena calcolate, nella funzione *read()* c'è una condizione non soddisfacibile nel caso true che porta a non poter raggiungere una parte di codice dove viene lanciata un' eccezione. Questo è determinato dal fatto che una chiamata al metodo *filechannel.read(),* precedente alla condizione, nel nostro caso non riesce a restituire un numero minore o uguale a zero in quanto l'unico parametro da noi controllato nei test che è possibile passargli è readBufferStartPosition che può essere un valore valido o negativo. Nel primo caso la chiamata del metodo va a buon fine e restituisce un valore positivo, nel secondo caso termina lanciando una IllegalArgumentException.

MUTATION TESTING BUFFEREDCHANNEL

Andiamo ora ad analizzare le mutazione generate dal framework pitest per i metodi *write()* e *read()*, visibili in Fig.6 e Fig.7, e come sono state uccise per aumentare l'adeguatezza della test suite. Partendo dal primo metodo, la **mutazione 126** sostituisce la somma con una sottrazione sbagliando così a salvare la nuova posizione corrente. Per ucciderla nel test *killMutation126()* si è dovuto ricorrere a due scritture consecutive in modo tale che, controllando la posizione nel buffer, ci si potesse accorgere della avvenuta mutazione. La **mutazione 129** sostituisce il >= con un > e quindi non avviene il flush sul file nel caso in cui i byte non persistenti siano pari al limite massimo di byte non persistenti (unpersistedBytesBound). Quindi *killMutation129()* controlla questo caso particolare scrivendo esattamente il numero di bytes pari all' unpersistedBytesBound e aspettandosi che il buffer sia svuotato sul file. Infine, la **mutazione 135**, nega una condizione che permette la scrittura forzata sul file in caso di superamento del limite di byte non persistenti. Il test *killMutation135()* scrive più byte di quanto sia il limite e controlla che non siano più presenti nel buffer. A questo punto, come è possibile vedere anche dalla Fig.6, tutte le mutazioni del metodo *write()* sono state uccise e possiamo quindi affermate che la sua MutationCoverage è pari a 1.
Passando al secondo metodo iniziamo con il controllare la **mutazione 240** che sostituisce la differenza con una somma. Questa modifica non veniva catturata dalla test suite precedente perché la *writeBufferStartPosition(),*ovvero la posizione iniziale di scrittura sul file, quando si entrava in quel branch era sempre pari a zero. Per questo è stato aggiunto un caso di test che arrivasse a scrivere sul file e allo stesso tempo rispettasse le condizioni per entrare nel branch. Infine la **mutazione 256** anche in questo caso andava a modificare una sottrazione con una addizione ma la *readBufferStartPosition* era pari a zero quindi non cambiava nulla. Con il caso di test *killMutation256()* si vanno a fare due letture consecutive per uccidere la mutazione. La **mutazione 266** invece modifica nella condizione dell'if un <= con un <. Questa condizione però non è soddisfacibile senza mutazione (come precedentemente spiegato), né tantomeno con la mutazione, di conseguenza è equivalente al SUT ed è una weak mutation in quanto segue esattamente lo stesso flusso di esecuzione. Per il metodo *read()* possiamo concludere che la MutationCoverage = $\frac{24}{26-1} = 0.96$

APACHE TAJO

BUILD LOCALE E CONFIGURAZIONE TRAVIS E SONARCLOUD

Una volta effettuato il clone del progetto Apache Tajo da github ed aver cancellato tutti i test presenti, per effettuare la build in locale sono state necessarie alcune operazioni. Per prima cosa si è dovuto configurare maven con una openjdk-8 e non superiori per problemi di compatibilità, dopodiché nella riga di comando per la compilazione si è dovuto aggiungere l'opzione *-Drat.skip* che permette di ignorare alcuni controlli. Questo framework chiamato rat, infatti, non permetteva il successo della build in quanto non riconosceva la licenza di alcuni file presenti in varie cartelle. Procedendo all'integrazione con Travis-CI e SonarCloud è stato scritto un file *".travis.yml"* riportato in Fig.8. Andiamo ad analizzarne le componenti:

1. Linguaggio da utilizzare Java
2. Disabilitati necessità di permessi da superuser
3. Campo addons necessario per inserire i dati per l'integrazione con SonarCloud
4. Campo install modificato rispetto a quello standard di Travis-CI per inserire l'opzione per ignorare il controllo delle licenze
5. Campo script per eseguire package e integrazione con SonarCloud

Per quanto riguarda i framework *Junit* e *Mockito* a differenza del progetto precedente si è dovuto integrarli nel pom principale del progetto, e anche con *Jacoco* e *Pitest* è stata necessaria l'integrazione e la configurazione dei pom in maniera analoga al progetto Bookkeeper (vedi Fig.2 e Fig.3).

CATEGORY PARTITION E PRIMI TEST PER TASKCOMPARATOR

In questa sezione si andrà ad analizzare la test suite minimale realizzata tramite category partition della classe TaskComparator che si occupa di ordinare, in base a diversi criteri, le istanze di tipo Task. In realtà TaskComparator, con il suo metodo *compare()*, è una inner class della classe JSPUtil. La classe Junit che si occupa del test è *TaskComparatorTest*. Per prima cosa il metodo *prepareTask(),* segnato con l'annotazione @Before e che viene eseguito prima di ogni test, si occupa di configurazioni necessarie ad instanziare i Task, mentre il metodo *createTask()* è quello che viene utilizzato per creare le istanze. Per testare il metodo *compare()* della classe TaskComparator si è dovuto sfruttare un metodo della classe JSPUtil che richiama quest'ultimo, ovvero il metodo *sortTasks()* :

```
public static void sortTasks(List<Task> tasks, String sortField, String sortOrder)
```

Notiamo che prende come input una lista di Task da ordinare, e due stringhe. In realtà osservando il codice possiamo notare come queste due stringhe non sono di qualsiasi tipo ma in realtà sono dei valori prefissati e quindi è possibile trattarle come se fossero delle enumerazioni. Viene lasciata la possibilità di inserire qualsiasi stringa in quanto in caso non sia conforme ad una delle possibili scelte, viene attuato un comportamento di default. Con queste precisazioni la divisione in classi di equivalenza quindi è la seguente:

1. List<Task> tasks: {valid, empty , null}. Lista dei task può essere valida (in questi test formata da 3 elementi) , vuota o null.
2. String order: {desc , asc , invalid, null} : possibilità di ordinamento (l'invalid viene reindirizzato nell'ordinamento desc)
3. String field: {id , host , runTime, startTime, invalid, null} : possibilità di campi rispetto a cui ordinare (il caso invalid e null vengono reindirizzati al campo id).

I casi di test ricavati dalla precedente divisione, con il metodo unidimensionale, sono stati divisi in vari metodi poiché ognuno, a seconda del campo scelto, ha un'implementazione differente. In *sortTaskIdTest()* vengono testati i seguenti casi di test :

```
"3,1,2,desc", // tasks = valid ; order = desc ; field = id
"3,1,2,asc", // tasks = valid ; order = asc ; field = id
"3,1,2," // tasks = valid ; order = invalid ; field = id
```

Il metodo *sortTaskHostTest()* non ha parametri in input perché gli host non sono inizializzati in questi Task (sono null). Vedremo in seguito anche il caso in cui sono effettivamente inizializzati.

```
//tasks = valid ; order = asc ; field = host
//tasks = valid ; order = desc ; field = host
```

Segue poi *sortTaskRunTimeTest()* che ordina i task in base alla loro durata :

```
"2,5,1,desc", //tasks = valid ; order = desc ; field = runTime
"2,5,1,asc" //tasks = valid ; order = asc ; field = runTime
```

Ed infine il metodo *sortTaskStartTimeTest()* che li ordina in base al tempo di inizio :

```
"2,5,1,desc", //tasks = valid ; order = desc ; field = startTime
"2,5,1,asc" //tasks = valid ; order = asc ; field = startTime
```

Gli ultimi due test rimasti, ovvero *sortTaskIdInvalidTasks()* e *sortTaskInvalidField()* si occupano di testare i casi invalid e null rispettivamente per la lista di Task e per il campo field :

```
// tasks = empty ; order = desc ; field = id
// tasks = null ;  order = desc ; field = id
//tasks = valid ; order = asc ; field = empty
// tasks = valid ; order = desc ; field = null
// tasks = valid ; order = desc ; field = invalid
```

Consapevolmente in questa prima analisi non ci si è strettamente attenuti al metodo unidimensionale, infatti ad esempio tutti i field compaiono almeno due volte sia in ordine decrescente che crescente. Si è deciso però comunque di aggiungerli subito in quanto la loro mancanza avrebbe indicato una evidente lacuna nel testing di uno solo dei due possibili ordinamenti.

AMPLIAMENTO TEST SUITE E ADEGUATEZZA TASKCOMPARATOR

A seguito della analisi fornita da SonarCloud sulla coverage (link : TaskComparator SonarCloud) sono stati implementati ulteriori casi di test per incrementare le varie metriche di adeguatezza. Innanzitutto è stato aggiunto il metodo *sortTaskHostMocked()* che utlizzando Mockito crea uno stub, quindi non inizializzandoli davvero,che restituisce una entry diversa da null per il campo host dei Task. In questo modo viene testata anche la funzionalità di TaskComparator nel caso in cui l'host non sia nullo. Questa scelta è stata effettuata per motivi di efficienza in quanto ci sarebbe stato un setup molto più complesso per istanziare dei veri host, ed anche perché gli host non sono di nostro interesse e non sono realmente utilizzati in questo scenario. Un'altra aggiunta effettuata è il metodo *sortTaskNullRuntime()* che ordina task che hanno l'istante di partenza pari a zero. Questo tipo di task infatti vengono trattati in modo speciale perché considerati sempre i primi a partire a prescindere da tutti gli altri, ed infatti nel metodo *compare()* di *TaskComparator* hanno un flusso di esecuzione a parte.

Andando ad analizzare i risultati di SonarCloud a seguito delle aggiunte effettuate, possiamo calcolare le seguenti metriche di adeguatezza per il metodo *compare()*:

1. Statement Coverage = 1 : Tutte le linee di codice sono state coperte
2. Branch Decision Coverage = 1 : Tutti i branch sono stati esplorati
3. Condition Coverage = 1 : Tutte le condizioni soddisfatte
4. Branch Condition Coverage = 1 : Tutte le condizioni ed i branch esplorati
5. Multiple Condition Coverage = 1 : Tutte le condizioni e le loro combinazioni soddisfatte

Stesse metriche sono ottenute per il metodo *sortTasks()* di JSPUtil, che è quello che viene utilizzato per poter eseguire *compare()*.

MUTATION TESTING TASKCOMPARATOR

A questo punto si è proceduto al mutation testing tramite pitest. Il risultato di queste operazioni sul metodo *compare()* è mostrato in Fig.9. Le uniche due mutazioni segnalate come non uccise sono proprio quelle mostrate nell'immagine, ovvero le **mutazioni 209 e 218** che sostituiscono al valore di ritorno pari a 1 il valore zero. Possiamo notare come queste in realtà siano equivalenti al SUT perché la classe Collections.sort() utilizza il valore del comparatore passatogli verificando se è >=0 oppure <0. Quindi passandogli 0 invece che 1 segue esattamente lo stesso flusso di esecuzione dando lo stesso risultato. Ci troviamo quindi di fronte a

due weak mutation. Passando quindi al calcolo della metrica ed otteniamo come risultato che MutationCoverage = $\frac{25}{27-2} = 1$ per il metodo *compare()* preso in considerazione.

CATEGORY PARTITION E PRIMI TEST PER FILEUTILS

In questa sezione passiamo ad analizzare la test suite minimale realizzata tramite category partition per i metodi della classe FileUtil. Partiamo con l'analizzare i metodi:

```
public static String readTextFile(File file) throws IOException
public static void writeTextToFile(String text, Path path) throws IOException
```

Servono semplicemente a scrivere e leggere una stringa da un file e prendono come input un path per il file e, nel caso della write, una stringa da scrivere. Questi due metodi sono stati testati in un unico metodo chiamato *writeAndReadTest()*, effettuando prima una scrittura e poi una lettura, e le classi di equivalenza individuate sono:

1. String path {valid, notExsisting, null} : path per raggiungere il file.
2. String textToWrite {len > 0 ; len = 0 ; null} : testo da scrivere all'interno del file

Una volta scritto qualcosa sul file quindi, si controlla che la stringa ritornata dalla lettura sia uguale a ciò che è stato scritto. Da questa category partition, utilizzando un approccio unidimensionale, i test effettuati per costruire una test suite minima sono:

```
"test.txt,prova", // path = valid ; textToWrite > 0
"test.txt,", // path = valid ; textToWrite = 0
"./newFolder/test.txt,prova", // path = notExisting ; textToWrite > 0
"test.txt,0", // path = valid ; textToWrite = null
"0,prova", // path = null ; textToWrite > 0
```

Analizziamo ora i test scritti per i metodi:

```
public static String readTextFromStream(InputStream inputStream) throws IOException
public static void writeTextToStream(String text, OutputStream outputStream) throws
IOException
```

Anche in questo caso i test servono a leggere e scrivere, solo che questa volta su uno stream che punta ad un file, invece che direttamente sul file. I test vengono effettuati dal metodo *writeAndReadFromStream()*, ma sia la category partition che i casi di test sviluppati a partire da questa sono uguale ai precedenti quindi si può fare riferimenti a quelli.
Passiamo ora ad un metodo che serve a mostrare il linguaggio umano un numero di bytes, ed è il seguente:

```
public static String humanReadableByteCount(long bytes, boolean si)
```

Applicando category partition otteniamo le seguenti suddivisioni:

1. long bytes : {< 0; = 0 ; 0 < bytes < 1000 ; >= 1000} : è stata fatta la scelta di fare una partizione anche per numeri minori o maggiori di mille per vedere come cambia l'unita di misura da B a kB.
2. boolean si : {true ; false} : booleano che indica se cambiare l'unita di misura a seconda dell'ordine di grandezza o lasciare in Byte.

I casi di test ricavati applicando un approccio unidimensionale sono i seguenti:

```
"-1,true,-1 B", // bytes < 0 ; si = true
"0,true,0 B", // bytes = 0 ; si = true
"1,false,1 B", // 0 < bytes < 1000 ; si = false
"1000,true,1.0 kB", //bytes >=1000 ; si = true
```

Per migliorare la comprensione del metodo sottoposto a test, a seguito degli input vengono lasciati anche i risultati attesi per ciascun caso di test.
Infine ci sono i metodi per la chiusura degli stream che sono:

```
public static void cleanup(Log log, java.io.Closeable... closeables)
```

```
public static void cleanupAndthrowIfFailed(java.io.Closeable... closeables) throws
IOException
```

Come si può leggere anche dalla documentazione all'interno del codice stesso, il primo ignora eventuali errori di chiusura scrivendo su un log l'errore, mentre il secondo in caso di errore lancia una IOException. Il metodo *cleanup()* viene testato da ***cleanupTest()*** e viene eseguita la seguente suddivisione:

1. Logger log { valid ; null }
2. Closable c { valid ; null}

Data la semplicità della suddivisione, è stato scelto in questo caso un approccio multidimensionale per la scelta dei seguenti casi di test:

```
"true,true", // log = valid ; c = valid
"true,false", // log = valid ; c = null
"false,true", // log = null ; c = valid
"false,false" // log = null ; c = null
```

Il secondo metodo viene invece testato da ***cleanupAndThrowTest()*** che semplicemente prende in input un'istanza valida di Closable e una null. In entrambi i casi però, non avendo un valore di ritorno dai metodi, per verificare la corretta esecuzione si è ricorso alla funzionalità *Mockito.verify()* sul metodo *close()* per controllare che effettivamente questo sia stato effettuato.

AMPLIAMENTO TEST SUITE E ADEGUATEZZA FILEUTIL

Una volta eseguiti i casi di test sopra descritti, tramite l'analisi dei risultati di SonarCloud (link FileUtil SonarCloud) si è ampliata la test suite in modo da incrementare la coverage dei test. Per prima cosa in ***humanReadableBytesTest()*** sono stati aggiunti i seguenti due casi di test:

```
"1000,false,1000 B", //bytes >=1000 ; si = false FOR COVERAGE
"1,true,1 B" // bytes > 0 ;si = true FOR COVERAGE
```

Sono serviti a coprire due condizioni particolari in cui si mostrano 1000 Byte scritti senza cambiare unità di misura e il contrario, ovvero 1 Byte cercando di cambiare unità di misura anche se non necessario. Ma le parti più importanti per l'aumento della coverage sono dovute a due nuovi metodi chiamati ***cleanupAndThrowExceptionTest()*** e ***cleanupExcpetionCloseTest()***. Sono due test che simulano un fallimento della chiamata *close(),* sull'oggetto Closable passato in input, grazie all'utilizzo di stub generati con il framework Mockito. Il primo metodo è un test per *cleanupAndthrowIfFailed(),*che notifica il fallimento tramite il lancio di IOException. Il secondo metodo invece è un test per *cleanup(),* che non notifica il fallimento della chiamata *close()* ma scrive su un log, se possibile, che la chiusura non è avvenuta. Per verificare l'effettivo funzionamento, anche in questo caso, si è dovuto ricorrere all'utilizzo della funzione *Mockito.verify()* sul metodo *close(),* in quanto *cleanup()* ignora i fallimenti e non restituisce nessun valore. A questo punto passiamo a calcolare le metriche di adeguatezza dei vari metodi, partendo da *readTextFile()* e *writeTextToFile():*

1. Statement Coverage = 1 : Tutte le linee di codice sono state coperte
2. Branch Decision Coverage = 1 : Tutti i branch sono stati esplorati
3. Condition Coverage = 1 : Tutte le condizioni soddisfatte
4. Branch Condition Coverage = 1 : Tutte le condizioni ed i branch esplorati
5. Multiple Condition Coverage = 1 : Tutte le condizioni possibili soddisfatte anche se presenti solo quelle semplici

Stessa analisi vale anche per *readTextFromStream()* e *writeTextToStream()*, ovvero tutte le metriche analizzate sono pari a 1.
Passiamo ora al calcolo delle metriche per *humanReadableByteCount()*:

1. Statement Coverage = 1 : Tutte le linee di codice sono state coperte
2. Branch Decision Coverage = 1 : Tutti i branch sono stati esplorati

3. Condition Coverage = $\frac{5}{6} = 0.83$: In una condizione composta, una condizione semplice non è coperta a causa di mancanza di documentazione e incomprensibilità della parte di codice.
4. Branch Condition Coverage = $\frac{5+1}{6+1} = 0.85$: Branch esplorati e condizione semplice non coperta
5. Multiple Condition Coverage = $\frac{6}{8} = 0.75$: Non tutte le possibili combinazioni di condizioni complesse coperte.

Per quanto riguarda *cleanup()* tutte le metriche sono pari ad 1. In *cleanupAndthrowIfFailed()* anche tutte le metriche sono pari ad 1, ma c'è una condizione non soddisfacibile. La condizione all'interno del catch infatti (ioe == null) non può essere mai falsa in quanto la variabile ioe viene inizializzata a null proprio all'inizio dell'esecuzione del metodo.

MUTATION TESTING FILEUTIL

Passiamo infine ad eseguire mutation testing sulla classe FileUtil i cui risultati riassunti vengono riportati in Fig.10. Analizziamo la **mutazione 62** che nega la condizione all'interno dell'if nel metodo *writeTextFile(),* che si occupa di creare il file in caso questo non esista. Questa mutazione non può essere uccisa in quanto equivalente al SUT. Infatti la chiamata !fs.exists(path.getParent()) venendo negata non comporta alcun problema in quanto non viene generata immediatamente la cartella con il relativo file in cui scrivere, ma viene creata successivamente dalla chiamata FSDataOutputStream out = fs.create(path). Producono quindi lo stesso output ma seguendo flussi diversi, per cui ci troviamo di fronte ad una strong mutation. L'altra mutazione da analizzare, che in questo caso però è stata uccisa, è la **mutazione 48** che rimuove completamente la chiamata ad *IOUtils.cleanup()* all'interno di un blocco *finally* del metodo *readTextFile().* L'unico modo per uccidere questa mutazione è controllare se effettivamente questa chiamata viene effettuata, utilizzando la funzione verify() di Mockito. Per fare ciò si è dovuto creare un'altra classe di test, chiamata *FileUtilMutationKillTest.* in quanto c'è bisogno di un Runner particolare chiamato *PowerMockRunner.* Questo runner particolare serve per poter utilizzare PowerMockito, un'estenzione di Mockito, che permette di utilizzare le varie funzionalità anche su metodi statici. Solo in questo modo quindi si è potuta verificare l'effettiva chiamata del metodo statico IOUtil.cleanup() e quindi uccidere la mutazione. Andando ad analizzare la metrica di copertura per le mutazioni, otteniamo come risultato che MutationCoverage = $\frac{26}{27-1} = 1.$

```
language: java
sudo: false
addons:
  sonarcloud:
    organization: "malavasiale"
    token: "**************************"

script:
  # the following command line builds the project, runs the tests with
  coverage and then execute the SonarCloud analysis

- mvn clean package sonar:sonar -Dsonar.projectKey=malavasiale_bookkeeper
```

Fig.1 File travis.yml per Apache Bookkeeper

```
…
<module>jacoco-tests-report-aggregator</module>
```

```
3.    </modules>
4.  …
5.  <plugin>
6.          <groupId>org.jacoco</groupId>
7.          <artifactId>jacoco-maven-plugin</artifactId>
8.          <version>${jacoco-maven-plugin.version}</version>
9.          <executions>
10.                 <execution>
11.                     <goals>
12.                         <goal>prepare-agent</goal>
13.                     </goals>
14.                 </execution>
15.                 <execution>
16.                     <id>report-aggregate</id>
17.                     <phase>prepare-package</phase>
18.                     <goals>
19.                         <goal>report</goal>
20.                     </goals>
21.                 </execution>
22.             </executions>
23.         </plugin>
24. …
```

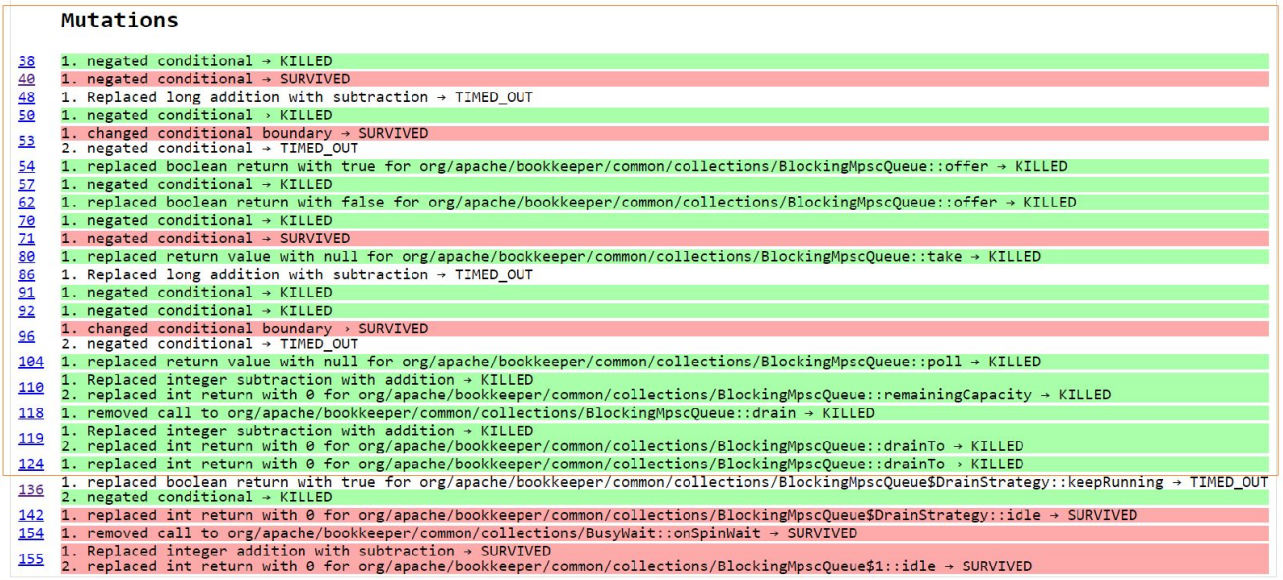
Fig.2 Esempio integrazione Jacoco in Bookkeeper

```
1.  <plugin>
2.     <groupId>org.pitest</groupId>
3.     <artifactId>pitest-maven</artifactId>
4.     <version>1.5.2</version>
5.     <configuration>
6.       <targetClasses>
7.
    <param>org.apache.bookkeeper.common.collections.BlockingMpscQueue*</param>
8.       </targetClasses>
9.       <targetTests>
10.        <param>mytests.BlockingMpscQueueTest</param>
11.     </targetTests>
12.   </configuration>
13. </plugin>
```

Fig.3 Esempio integrazione Pitest su bookkeeper

```
1.  public ConcurrentCircularArrayQueue(int capacity)
2.      {
3.          int actualCapacity = Pow2.roundToPowerOfTwo(capacity);
4.          mask = actualCapacity - 1;
5.          buffer = CircularArrayOffsetCalculator.allocate(actualCapacity);
6.      }
7.
```

Fig.4 Costruttore classe BlockingMpscQueue



Mutations

38	1. negated conditional → KILLED
40	1. negated conditional → SURVIVED
48	1. Replaced long addition with subtraction → TIMED_OUT
50	1. negated conditional → KILLED
53	1. changed conditional boundary → SURVIVED 2. negated conditional → TIMED_OUT
54	1. replaced boolean return with true for org/apache/bookkeeper/common/collections/BlockingMpscQueue::offer → KILLED
57	1. negated conditional → KILLED
62	1. replaced boolean return with false for org/apache/bookkeeper/common/collections/BlockingMpscQueue::offer → KILLED
70	1. negated conditional → KILLED
71	1. negated conditional → SURVIVED
80	1. replaced return value with null for org/apache/bookkeeper/common/collections/BlockingMpscQueue::take → KILLED
86	1. Replaced long addition with subtraction → TIMED_OUT
91	1. negated conditional → KILLED
92	1. negated conditional → KILLED
96	1. changed conditional boundary → SURVIVED 2. negated conditional → TIMED_OUT
104	1. replaced return value with null for org/apache/bookkeeper/common/collections/BlockingMpscQueue::poll → KILLED
110	1. Replaced integer subtraction with addition → KILLED 2. replaced int return with 0 for org/apache/bookkeeper/common/collections/BlockingMpscQueue::remainingCapacity → KILLED
118	1. removed call to org/apache/bookkeeper/common/collections/BlockingMpscQueue::drain → KILLED
119	1. Replaced integer subtraction with addition → KILLED 2. replaced int return with 0 for org/apache/bookkeeper/common/collections/BlockingMpscQueue::drainTo → KILLED
124	1. replaced int return with 0 for org/apache/bookkeeper/common/collections/BlockingMpscQueue::drainTo → KILLED
136	1. replaced boolean return with true for org/apache/bookkeeper/common/collections/BlockingMpscQueue$DrainStrategy::keepRunning → TIMED_OUT 2. negated conditional → KILLED
142	1. replaced int return with 0 for org/apache/bookkeeper/common/collections/BlockingMpscQueue$DrainStrategy::idle → SURVIVED
154	1. removed call to org/apache/bookkeeper/common/collections/BusyWait::onSpinWait → SURVIVED
155	1. Replaced integer addition with subtraction → SURVIVED 2. replaced int return with 0 for org/apache/bookkeeper/common/collections/BlockingMpscQueue$1::idle → SURVIVED

Fig.5 Mutazioni BlockingMpscQueue

```
110      public void write(ByteBuf src) throws IOException {
111          int copied = 0;
112          boolean shouldForceWrite = false;
113          synchronized (this) {
114              int len = src.readableBytes();
115 2            while (copied < len) {
116 1                int bytesToCopy = Math.min(src.readableBytes() - copied, writeBuffer.writableBytes());
117 1                writeBuffer.writeBytes(src, src.readerIndex() + copied, bytesToCopy);
118 1                copied += bytesToCopy;
119
120                  // if we have run out of buffer space, we should flush to the
121                  // file
122 1                if (!writeBuffer.isWritable()) {
123 1                    flush();
124                  }
125              }
126 1            position += copied;
127 1            if (doRegularFlushes) {
128                  unpersistedBytes.addAndGet(copied);
129 2                if (unpersistedBytes.get() >= unpersistedBytesBound) {
130 1                    flush();
131                      shouldForceWrite = true;
132                  }
133              }
134          }
135 1        if (shouldForceWrite) {
136              forceWrite(false);
```

Fig.6 Mutazioni write() BufferedChannel

```
234      @Override
235      public synchronized int read(ByteBuf dest, long pos, int length) throws IOException {
236          long prevPos = pos;
237 2        while (length > 0) {
238              // check if it is in the write buffer
239 3            if (writeBuffer != null && writeBufferStartPosition.get() <= pos) {
240 1              int positionInBuffer = (int) (pos - writeBufferStartPosition.get());
241 1              int bytesToCopy = Math.min(writeBuffer.writerIndex() - positionInBuffer, dest.writableBytes());
242
243 1              if (bytesToCopy == 0) {
244                      throw new IOException("Read past EOF");
245                  }
246
247                  dest.writeBytes(writeBuffer, positionInBuffer, bytesToCopy);
248 1                pos += bytesToCopy;
249 1                length -= bytesToCopy;
250 3            } else if (writeBuffer == null && writeBufferStartPosition.get() <= pos) {
251                  // here we reach the end
252                  break;
253                  // first check if there is anything we can grab from the readBuffer
254 5            } else if (readBufferStartPosition <= pos && pos < readBufferStartPosition + readBuffer.writerIndex()) {
255 1                int positionInBuffer = (int) (pos - readBufferStartPosition);
256 1                int bytesToCopy = Math.min(readBuffer.writerIndex() - positionInBuffer, dest.writableBytes());
257                  dest.writeBytes(readBuffer, positionInBuffer, bytesToCopy);
258 1                pos += bytesToCopy;
259 1                length -= bytesToCopy;
260                  // let's read it
261              } else {
262                  readBufferStartPosition = pos;
263
264                  int readBytes = fileChannel.read(readBuffer.internalNioBuffer(0, readCapacity),
265                          readBufferStartPosition);
266 2                if (readBytes <= 0) {
267                      throw new IOException("Reading from filechannel returned a non-positive value. Short read.");
268                  }
269                  readBuffer.writerIndex(readBytes);
270              }
271          }
272 2        return (int) (pos - prevPos);
273      }
```

Fig.7 Mutazioni read() BufferedChannel

```
1.  language: java
2.  sudo: false
3.  jdk:
4.      -openjdk8
5.  addons:
6.    sonarcloud:
7.      organization: "malavasiale"
8.      token: "**************************"
9.  install:
10.     - mvn clean install -Drat.skip=true
11. script:
12.   # the following command line builds the project, runs the tests with
    coverage and then execute the SonarCloud analysis
13.     - mvn clean package sonar:sonar -Dsonar.projectKey=malavasiale_tajo
```

Fig.8 File travis.yml per Apache Tajo

```
192      public int compare(Task task, Task task2) {
193 1      if(asc) {
194 1        if("id".equals(sortField)) {
195 1          return task.getId().compareTo(task2.getId());
196 1        } else if("host".equals(sortField)) {
197 1          String host1 = task.getSucceededWorker() == null ? "-" : task.getSucceededWorker().getHost();
198 1          String host2 = task2.getSucceededWorker() == null ? "-" : task2.getSucceededWorker().getHost();
199 1          return host1.compareTo(host2);
200 1        } else if("runTime".equals(sortField)) {
201 1          return compareLong(task.getRunningTime(), task2.getRunningTime());
202 1        } else if("startTime".equals(sortField)) {
203 1          return compareLong(task.getLaunchTime(), task2.getLaunchTime());
204          } else {
205 1          return task.getId().compareTo(task2.getId());
206          }
207        } else {
208 1        if("id".equals(sortField)) {
209 1          return task2.getId().compareTo(task.getId());
210 1        } else if("host".equals(sortField)) {
211 1          String host1 = task.getSucceededWorker() == null ? "-" : task.getSucceededWorker().getHost();
212 1          String host2 = task2.getSucceededWorker() == null ? "-" : task2.getSucceededWorker().getHost();
213 1          return host2.compareTo(host1);
214 1        } else if("runTime".equals(sortField)) {
215 1          if(task2.getLaunchTime() == 0) {
216 1            return -1;
217 1          } else if(task.getLaunchTime() == 0) {
218 1            return 1;
219            }
220 1          return compareLong(task2.getRunningTime(), task.getRunningTime());
221 1        } else if("startTime".equals(sortField)) {
222 1          return compareLong(task2.getLaunchTime(), task.getLaunchTime());
223          } else {
224 1          return task2.getId().compareTo(task.getId());
225          }
226        }
227      }
228    }
```

Fig.9 Mutazioni compare() TaskComparator

Mutations

42	1. negated conditional → KILLED
48	1. removed call to org/apache/hadoop/io/IOUtils::cleanup → KILLED
50	1. replaced return value with "" for org/apache/tajo/util/FileUtil::readTextFile → KILLED
62	1. negated conditional → SURVIVED
66	1. removed call to org/apache/hadoop/fs/FSDataOutputStream::write → KILLED
67	1. removed call to org/apache/hadoop/fs/FSDataOutputStream::close → KILLED
76	1. negated conditional → KILLED
80	1. replaced return value with "" for org/apache/tajo/util/FileUtil::readTextFromStream → KILLED
82	1. removed call to org/apache/hadoop/io/IOUtils::closeStream → KILLED
88	1. removed call to java/io/OutputStream::write → KILLED
90	1. removed call to org/apache/hadoop/io/IOUtils::closeStream → KILLED
95	1. negated conditional → KILLED
96	1. changed conditional boundary → KILLED 2. replaced return value with "" for org/apache/tajo/util/FileUtil::humanReadableByteCount → KILLED 3. negated conditional → KILLED
97	1. Replaced double division with multiplication → KILLED
98	1. Replaced integer subtraction with addition → KILLED 2. negated conditional → KILLED 3. negated conditional → KILLED
99	1. replaced return value with "" for org/apache/tajo/util/FileUtil::humanReadableByteCount → KILLED 2. Replaced double division with multiplication → KILLED
112	1. negated conditional → KILLED
114	1. removed call to java/io/Closeable::close → KILLED
132	1. negated conditional → KILLED
134	1. removed call to java/io/Closeable::close → KILLED
136	1. negated conditional → KILLED
141	1. negated conditional → KILLED

Fig.10 Mutazioni FileUtil