ISW2 Project A.A. 2019-2020

Andrea Graziani (0273395)
andrea.graziani93@outlook.it
Università degli Studi di Roma "Tor Vergata"
Rome, Italy

KEYWORDS

Testing, Equivalence Class Partitioning, Boundary Value Analysis, Mutation Testing

1 INTRODUCTION

In this report we will describe all results and issues about the utilization of several *testing techniques* involving following open source projects:

Apache BookKeeper™ Which is a scalable, fault tolerant and low latency storage service[1].

Apache OpenJPA™ An implementation of the Java Persistence API specification[2].

To be more precise, the aim of this report is to analyse 2 classes of each aforementioned project, attempting both to find if there are any errors in it and to increase our confidence in the correct functioning of the **software under test** (**SUT**), although the completeness of our testing does not necessarily demonstrate that these classes are error free.

1.1 Class selection

To select classes for our testing activities, we have mainly relied on results elaborated by our **defect prediction model**, developed using **Weka** [3] machine learning software, using a *random forests* classifier without *sampling* or *feature selection*.

Particularly, we have focused on classes marked as *defective* (or *buggy*) by our predictor because it is likely that they exhibit a defect observable by our test sets. As known, this approach is able to *reduce the cost of testing*, by letting to focus on specific classes, ignoring stable ones [5].

In particular, for our testing activities involving Apache Book-Keeper™, we have choose following classes:

DefaultEnsemblePlacementPolicy [4] Because it has been marked as defective by our prediction model and, as we will see later, our testing activity has confirmed that prediction.

DiskChecker [5] Although it has not been marked as defective, we have choose this class due to its relative high LOC value (294), high number of revisions (11) and high number of authors (9). Despite its presumed stability, our testing activity has found several defects inside that class.

1.2 Testing technique

In order to build our test set, we have adopted a testing technique called *equivalence class partitioning*, according to which the domain of possible input data for each input data element is divided into **equivalence classes**; an equivalence class is a set of data values that the tester assumes are processed in the same way by the test object [6].

Equivalence class definition was been mainly based on *specifications* of our test objects using, therefore, a *black-box* approach; however, sometimes, due to lacking of detailed specifications, we have performed our analysis looking the code too, using conversely a **while-box** approach.

For any equivalence classes is important to find a **representative**. As known, testing one representative of the equivalence class is considered sufficient because it is assumed that for any other input value of the same equivalence class, the test object will show the same reaction or behaviour [6]. Besides equivalence classes for *correct* input, those for *incorrect* input values must be tested as well.

To guarantee that all test object reactions are triggered, we have combined representative values using following rules:

(1) The representative value of all valid equivalence classes have been combined to test cases (*valid/positive test case*), meaning that all possible combinations of valid equivalence classes will be covered.

(2) The representative value of an invalid equivalence class have been combined only with representatives of other valid equivalence classes (*invalid/negative test case*).

Moreover, we have adopted following guidelines[6]:

(1) Test cases including boundary values combinations are preferred.

(2) Every representative of an equivalence class appears in at least one test case.

(3) Representatives of invalid equivalence classes should not be combined with representatives of other invalid equivalence classes.

2 APACHE BOOKKEEPER™

2.1 DefaultEnsemblePlacementPolicy class testing analysis

What are responsibilities of DefaultEnsemblePlacementPolicy class? What meant by ensemble? And what for placement policy?

According to bookkeeper specifications[1], an **ensemble** represents a group of **bookies** storing **entries**. To be more precise, according to bookkeeper's nomenclature, a **bookie** is an individual storage server while **entries** represent stored data, therefore an ensemble of size E represents simply a group of storage servers.

The aim of this design is to guarantee **consistency** in an ensemble of bookies of all stored data exploiting a **quorum-based replicated-write** protocol. As known, to support replicated writes at multiple replicas of a file, a client must first contact at least half the servers plus one (a majority) and get them to agree to do the

---

[1]https://github.com/apache/bookkeeper
[2]https://github.com/apache/openjpa
[3]https://www.cs.waikato.ac.nz/~ml/weka/
[4]org.apache.bookkeeper.client.DefaultEnsemblePlacementPolicy
[5]org.apache.bookkeeper.util.DiskChecker

update[7]. Technically, in order to modify a file, a client needs to assemble the so called **write quorum**, that is an arbitrary collection of servers which must be more than the half of all available servers[7]. Therefore, using bookkeeper's nomenclature, the size of write quorum Q_w represents the number of bookies where each entry is written.

According to bookKeeper protocol[3], the following invariant must hold:

$$E \geqslant Q_w \geqslant Q_a \tag{1}$$

In other word, the ensemble size E must be larger than the write quorum size Q_w, which must in turn be larger than the so called **ack quorum size** Q_a, which represents, instead, the number of nodes an entry must be acknowledged on.

BookKeeper uses several algorithms to selects a number of bookies from a cluster to build an ensemble compliant to above specifications, some of which are capable to exploit several network topology proprieties too.

According to BookKeeper design, the implementations of these algorithm must be compliant to `EnsemblePlacementPolicy` interface [4]; in other words, any implementation must respect a specific *contract*, established by aforementioned interface, which covers aspects related to initialization and bookie selection for data placement and reads[4].

Currently there are 3 implementations available by default. They are:

- `DefaultEnsemblePlacementPolicy`
- `RackawareEnsemblePlacementPolicy`
- `RegionAwareEnsemblePlacementPolicy`

In particular, `DefaultEnsemblePlacementPolicy` class encapsulates the simplest algorithm for bookie selection for ensemble creation because it, simply, picks bookies randomly in order to build an ensemble.

2.1.1 `initialize`.

According to documentation, after the creation of an `Default-EnsemblePlacementPolicy` object's instance, bookKeeper client have to call `initialize` method, which signature is reported in **listing 1**, in order to initialize the placement policy.

Observing method's signature and documentation, we note that several complex resources, passed as parameter, are necessary to invoke this method, however, observing method's implementation, only one of them is effectively used, simplifying our testing activity.

In fact, only `conf` parameter, representing a `ClientConfiguration` object, used to contain client configuration necessary for bookKeeper client construction, is effectively used.

Moreover, only following methods, belonging `ClientConfiguration` class, using `conf` parameter, are invoked in `initialize` method:

- `getDiskWeightBasedPlacementEnabled`
- `getBookieMaxWeightMultipleForWeightBasedPlacement`

Therefore, in order to test `initialize` method, we have choose to use a **mock**. As known, a mock is a special-purpose replacement class that mimic behaviour of real objects in controlled ways, for example returning values to the calling component. As known, the use of mock is very useful during testing activity of incomplete

portions of software because you have not to to wait for their completion to start your test activity.

Oblivious, we have use **Mockito framework**[6] to manage `conf` parameter's return values.

All valid and invalid equivalence classes, indicated as *vEC* and *iEC* respectively, are reported in **table 1**.

Listing 1: Signature of method `initialize`

```
EnsemblePlacementPolicy initialize(ClientConfiguration conf,
                    Optional<DNSToSwitchMapping> optionalDnsResolver,
                    HashedWheelTimer hashedWheelTimer,
                    FeatureProvider featureProvider,
                    StatsLogger statsLogger)
```

2.1.2 `newEnsemble`.

Documentation [4] reports that `newEnsemble` method is used to build an ensemble made up of several bookies. Method signature, shown below in **listing 2**, reports that it takes up to 5 parameters and return an ensemble which is represented by a `List<BookieSocket-Address>` object.

Listing 2: Signature of method `newEnsemble`

```
List<BookieSocketAddress> newEnsemble(int ensembleSize,
                    int writeQuorumSize,
                    int ackQuorumSize,
                    Map<String,byte[]> customMetadata,
                    Set<BookieSocketAddress> excludeBookies)
            throws BKException.BKNotEnoughBookiesException
```

The meaning of all parameter is:

`ensembleSize` represents the ensemble size.

`writeQuorumSize` the value of write quorum size.

`ackQuorumSize` the value of ack quorum Size.

`customMetadata` user meta-data.

`excludeBookies` a collection of bookies that should not be considered as targets for the new ensemble.

Is very important to precise that, in order to build an ensemble, bookies are picked up from a set called `knownBookies`, which can be populated invoking `onClusterChanged` method (which we will describe following).

Moreover, please note that we will use K to represent the set of `BookieSocketAddress` objects of `knownBookies`, while E, conversely, will be used to refer to the set of `BookieSocketAddress` objects belonging to `excludeBookies`.

All valid and invalid equivalence classes are reported in **table 2**.

For this method, we have developed $2 \times 1 \times 1 \times 1 \times 2 = 4$ valid test cases, by combining the representatives of the equivalence classes, and $2 + 1 + 1 + 3 = 7$ negative test cases, by separately testing representatives of every invalid class, with a total of 11 test cases. However, every test case was been executed twice because, as said previously, two different forms of initialization of `DefaultEnsemblePlacementPolicy` object exist (see `initialize` method); therefore, we have executed a total of 22 tests; of these, 8 failed.

2.1.3 `onClusterChanged`.

`onClusterChanged` method is used to update the view of the cluster, that is to specify what bookies are available as *writeable* and what bookies are available as *read-only*; this operation is necessary to populate, or update, `knownBookies` set which, as already said, is used to pick up bookies during `newEnsemble` invocation. According

---

[6]https://site.mockito.org/

Table 1: Equivalence classes and representatives of `initialize` method

Parameter		Equivalence Classes	Representatives
conf	vEC_1	A not-null `ClientConfiguration` object which: • `conf.getDiskWeightBasedPlacementEnabled() = true`	*(see the code)*
	vEC_1	A not-null `ClientConfiguration` object which: • `conf.getDiskWeightBasedPlacementEnabled() = false`	*(see the code)*
	iEC_1	null object	`null`
optionalDnsResolver	vEC_1	Any value. `optionalDnsResolver` is never used or accessed by the class.	`null`
hashedWheelTimer	vEC_1	Any value. `hashedWheelTimer` is never used or accessed by the class.	`null`
featureProvider	vEC_1	Any value. `featureProvider` is never used or accessed by the class.	`null`
statsLogger	vEC_1	Any value. `statsLogger` is never used or accessed by the class.	`null`

Table 2: Equivalence classes and representatives of `newEnsemble` method

Parameter		Equivalence Classes	Representatives
ensembleSize	vEC_1	`ensembleSize` $= 0$	0
	vEC_2	$0 <$ `ensembleSize` $\leq |K| - |E|$	$|K| - |E|$
	iEC_1	`ensembleSize` < 0	-1
	iEC_2	`ensembleSize` $> |K| - |E|$	$|K| - |E| + 1$
writeQuorumSize	vEC_1	`writeQuorumSize` $\leq$ `ensembleSize`	`ensembleSize`
	iEC_1	`writeQuorumSize` $>$ `ensembleSize`	`ensembleSize + 1`
ackQuorumSize	vEC_1	`ackQuorumSize` $\leq$ `writeQuorumSize`	`writeQuorumSize`
	iEC_1	`ackQuorumSize` $>$ `writeQuorumSize`	`ackQuorumSize + 1`
customMetadata	vEC_1	Any value. `customMetadata` is never used or accessed by the class.	`null`
excludeBookies	vEC_1	A not-null `Set<BookieSocketAddress>` object where: • $|E| = 0$	`new HashSet<>()`
	vEC_2	A not-null `Set<BookieSocketAddress>` object where: • $0 < |E| < |K|$ • $E \subset K$ • $|K| - |E| \geq$ `ensembleSize`	*(see the code)*
	iEC_1	A not-null `Set<BookieSocketAddress>` object where: • $|E| > 0$ • $\exists x \in E : x \notin K$	*(see the code)*
	iEC_2	A not-null `Set<BookieSocketAddress>` object where: • $|E| > 0$ • $\exists x \in E : x =$ `null`	*(see the code)*
	iEC_3	null object	`null`

to documentation, `onClusterChanged` should be invoked when any changes happen in the cluster, returning a list of *failed* (or *dead*) bookies during this cluster change.

Signature is reported in **listing 3**.

Listing 3: Signature of method `onClusterChanged`

```
Set<BookieSocketAddress> onClusterChanged(Set<BookieSocketAddress> writableBookies,
                                          Set<BookieSocketAddress> readOnlyBookies)
```

All valid and invalid equivalence classes are reported in **table 3**.

Combining the representatives of the equivalence classes, we have developed are $2 \times 2 = 4$ valid test cases and $3 + 3 = 6$ negative test cases, by separately testing representatives of every invalid class. Therefore, we have a 10 distinct tests and, since we have to run each test twice, we have a total of 20 test cases; of these, 8 failed.

2.1.4 updateBookieInfo.
According to `DefaultEnsemblePlacementPolicy` interface's documentation [4], `updateBookieInfo` is used to update bookie info

details, taking only one input parameter, a `Map<BookieSocketAddress, BookieInfo>` object.

Please note that we will use K to represent the set of the *keys* (or *indexes*) of the dictionary `bookieInfoMap`, that is the set of all `BookieSocketAddress` objects used as indexes. Conversely, V will be used to refer to the set of the *values* of the dictionary, made up of `BookieInfo` objects.

Signature is reported in **listing 4**, while equivalence classes are shown in **table 4**. In this case, we have developed are 2 valid test cases and 4 negative test cases. Therefore, we have 6 distinct tests and, since we have to run each test twice, we have a total of 12 test cases; of these, 3 failed.

Listing 4: Signature of method `updateBookieInfo`

```
void updateBookieInfo(Map<BookieSocketAddress,BookieInfo> bookieInfoMap)
```

2.1.5 replaceBookie.
Documentation reports that `replaceBookie` is choose randomly a

Table 3: Equivalence classes and representatives of `onClusterChanged` method

Parameter		Equivalence Classes	Representatives
writableBookies	vEC_1	An not-null Set<BookieSocketAddress> object where: • writableBookies.size() = 0	new HashSet<>()
	vEC_2	A not-null Set<BookieSocketAddress> object where: • writableBookies.size() > 0 • writableBookies ∩ readOnlyBookies = ∅	*(see the code)*
	iEC_1	A not-null Set<BookieSocketAddress> object where: • writableBookies.size() > 0 • $\exists x \in$ writableBookies : $x \in$ readOnlyBookies	*(see the code)*
	iEC_2	A not-null Set<BookieSocketAddress> object where: • writableBookies.size() > 0 • $\exists x \in$ writableBookies : $x =$ null	*(see the code)*
	iEC_3	null object	null
readOnlyBookies	vEC_1	An not-null Set<BookieSocketAddress> object where: • readOnlyBookies.size() = 0	new HashSet<>()
	vEC_2	A not-null Set<BookieSocketAddress> object where: • readOnlyBookies.size() > 0 • writableBookies ∩ readOnlyBookies = ∅	*(see the code)*
	iEC_1	A not-null Set<BookieSocketAddress> object where: • readOnlyBookies.size() > 0 • $\exists x \in$ readOnlyBookies : $x \in$ writableBookies	*(see the code)*
	iEC_2	A not-null Set<BookieSocketAddress> object where: • readOnlyBookies.size() > 0 • $\exists x \in$ readOnlyBookies : $x =$ null	*(see the code)*
	iEC_3	null object	null

Table 4: Equivalence classes and representatives of `updateBookieInfo` method

Parameter		Equivalence Classes	Representatives
bookieInfoMap	vEC_1:	A not-null Map<BookieSocketAddress, BookieInfo> object where: • bookieInfoMap.size() = 0	*(see the code)*
	vEC_2:	A not-null Map<BookieSocketAddress, BookieInfo> object where: • bookieInfoMap.size() > 0 • $\forall (x, y) \in K \times V \Rightarrow x \neq null, y \neq null$	*(see the code)*
	iEC_1:	A not-null Map<BookieSocketAddress, BookieInfo> object where: • bookieInfoMap.size() > 0 • $\exists (x, y) \in K \times V : x = null, y \neq null$	*(see the code)*
	iEC_2:	A not-null Map<BookieSocketAddress, BookieInfo> object where: • bookieInfoMap.size() > 0 • $\exists (x, y) \in K \times V : x \neq null, y = null$	*(see the code)*
	iEC_3:	A not-null Map<BookieSocketAddress, BookieInfo> object where: • bookieInfoMap.size() > 0 • $\exists (x, y) \in K \times V : x = null, y = null$	*(see the code)*
	iEC_4:	null object.	null

bookie form `knownBookies` set, if available, in order to replace it with a bookie passed as parameter.

Observing method's implementation and its signature, the latter reported in **listing 5**, it is easy to notice a similarity with `newEnsamble` the method, described previously. Despite very little differences, equivalence classes of both methods are, practically, the same; please see **table 5**.

Listing 5: Signature of method `replaceBookie`

```
BookieSocketAddress replaceBookie(int ensembleSize,
                                  int writeQuorumSize,
                                  int ackQuorumSize,
                                  Map<String,byte[]> customMetadata,
                                  Set<BookieSocketAddress> currentEnsemble,
                                  BookieSocketAddress bookieToReplace,
                                  Set<BookieSocketAddress> excludeBookies)
                           throws BKException.BKNotEnoughBookiesException
```

Table 5: Equivalence classes and representatives of `replaceBookie` method

Parameter		Equivalence Classes	Representatives
`ensembleSize`		See **table 2**	
`writeQuorumSize`		See **table 2**	
`ackQuorumSize`		See **table 2**	
`customMetadata`		See **table 2**	
`currentEnsemble`	vEC_1	A not-null List<BookieSocketAddress> object where: • currentEnsemble.size() > 0 • $\forall x \in$ currentEnsemble $\Rightarrow x \in K$ • $\forall x \in$ currentEnsemble $\Rightarrow x \neq$ null	*(see the code)*
	iEC_1	A not-null List<BookieSocketAddress> object where: • currentEnsemble.size() = 0	`new ArrayList<>()`
	iEC_2	A not-null List<BookieSocketAddress> object where: • currentEnsemble.size() > 0 • $\forall x \in$ currentEnsemble $\Rightarrow x \neq$ null • $\exists x \in$ currentEnsemble : $x \notin K$	*(see the code)*
	iEC_3	A not-null List<BookieSocketAddress> object where: • currentEnsemble.size() > 0 • $\exists x \in$ currentEnsemble : $x =$ null	*(see the code)*
	iEC_4	null object	`null`
`bookieToReplace`	vEC_1	A valid BookieSocketAddress object where: • bookieToReplace $\in$ currentEnsemble • bookieToReplace $\neq$ null	*(see the code)*
	iEC_1	A valid BookieSocketAddress object where: • bookieToReplace $\notin$ currentEnsemble • bookieToReplace $\neq$ null	*(see the code)*
	iEC_2	null object	`null`
`excludeBookies`		See **table 2**	

2.1.6 `Other methods`.
To be precise, `DefaultEnsemblePlacementPolicy` class contains many other methods not analysed in this report because either they does nothing or they return always the same value regardless input parameters.

Those methods are:

`uninitalize` This method does nothing.
`registerSlowBookie` This method does nothing too.
`isEnsembleAdheringToPlacementPolicy` Return always PlacementPolicyAdherence.MEETS_STRICT regardless input parameters.
`reorderReadSequence` Returns always a `WriteSet` object which is passed as parameter.

2.2 `DiskChecker` class testing analysis

According to BookKeeper's documentation, `DiskChecker` class is used to provide several utility functions for checking disk problems, managing all directories belonging to **ledgers**, which represents the basic unit of storage in BookKeeper[1].

Specifications establishes that, for each ledger directory, is possible to specify a maximum disk space D which can be used and a, so-called, warning threshold D_w for disk usage[2]. In particular, the following invariant must hold:

$$0 < \quad D \quad < 1$$
$$D_w \leq \quad D \tag{2}$$

However, for both thresholds, BookKeeper's specifications establish a default value, which is equal to 0.95.

Using out test set T_1, which is retrievable from commit `bdf2fc28d2`[7], our testing activity reports that up to 5 unit tests have failed, out of a total of 31, revealing the presence of several bugs, despite its stability according to our prediction model. In following subsections, we will describe how we have build our test sets.

2.2.1 `DiskChecker`.
Obliviously, that method represents the constructor of `DiskChecker` class and takes up to 2 parameters:

`threshold` previously indicated as D
`warnThreshold` previously indicated as D_w

All valid and invalid equivalence classes, which definition is clearly based on specifications described before, are reported in **table 6**, while method signature is reported in **listing 6**.

We have developed $1 \times 1 = 1$ valid test cases (by combining the representatives of the valid equivalence classes) and $1 + 3 = 4$ negative test cases. Since test cases belonging to both iEC_3 of `threshold` parameter and iEC_1 of `warnThreshold` overlap, in total 4 test cases result from the 6 equivalence classes.

Equivalence classes are shown **table 6**. while test cases are reported in **table 7**.

Listing 6: Signature of method `DiskChecker`

```
public DiskChecker(float threshold, float warnThreshold)
```

---
[7]https://github.com/AndreaG93/ISW2-bookkeeper/tree/
bdf2fc28d216b0adf9e467cfb51c4f334e84d5c9

2.2.2 `checkDir`.

`checkDir` method, taking only one `File` abject as parameter, is used to perform several checks involving a directory, verifying disk usage, write and read permissions or the exceeding of the disk usage threshold.

It returns the disk usage fraction usage, defined as following:

$$\text{usage} = 1 - \frac{\text{usableSpace}}{\text{totalSpace}} \quad (3)$$

All valid and invalid equivalence classes are reported in table **table 8**, while method signature is reported in **listing 7**. Test cases are described in **table 9**.

During equivalence class definitions, it was assumed that Book-Keeper's user is *not* `root` user.

Listing 7: Signature of method `checkDir`

```
public float checkDir(File dir) throws DiskErrorException
```

2.2.3 `getTotalDiskUsage`.

`getTotalDiskUsage` is used to compute disk usage fraction, already defined in **3**, taking as input a list of directories.

All valid and invalid equivalence classes are reported in table **table 10**, while method signature is reported in **listing 8**

Listing 8: Signature of method `getTotalDiskUsage`

```
public float getTotalDiskUsage(List<File> dirs) throws IOException
```

2.2.4 `getTotalDiskSpace` – `getTotalFreeSpace`.

Observing both the implementations and the signatures of these last methods, called `getTotalDiskSpace` and `getTotalFreeSpace`, is easy to understand their meaning and aim, however, if we look at the code carefully, we will note several defects that must be taken into account during test activities:

(1) Unexpectedly, comments of both methods are the same, although they perform different activities; in other words, methods comments "*lie*", because they do not describe what the functions truly do.
The only way to build our test set is to observe method implementations ignoring comments, adopting a pure white-box approach.
This consideration is true for `getTotalDiskSpace`.
For `getTotalFreeSpace` method, comment is just imprecise.

(2) `getTotalFreeSpace` method signature "*lies*" too. In fact, although we would expect that the method returns "*total free space*" of a set of directories, observing the implementation, is easy to understand that it returns, instead, the *total usable space* of a set of directories.
Since, usable space and free space have different meaning, this erroneous signature is the reason according to which many of our test cases failed.

An other defect, not strictly related to our testing activities, is that methods implementations are the same (except one line). In other words, this is a case of code duplication.

Although some differences (an empty `List<File>` object is a valid input), equivalence classes of both these methods are the same of the `getTotalDiskUsage` method, already described in **10**.

JaCoCo [8]

2.3 Adequacy Criteria

2.3.1 *Statement coverage.*

Our test set T is able to cover up to 85 statements out of a total of 89, reaching, according to Sonarcloud report, a *statement coverage* equal to **0,955 (95,5%)**.

However we believe that `DiskChecker` class contains several statements that we can consider as **unreachable** because they fall on an *infeasible path*, that is a path that would never be reached by our test set T with any type of input data; to be more precise, we believe that `setDiskSpaceThreshold(float, float)` method contains up to 4 unreachable statements.

Generally, in order to identify unreachable statements, drawing the flow-graph of the developed code and finding out the path that would never be reached is required, however we can consider as unreachable all statements of `setDiskSpaceThreshold(float, float)` method owing to following reasons:

(1) `setDiskSpaceThreshold(float, float)` method is **never used** in the project, that is it is never called by any other method.

(2) `setDiskSpaceThreshold(float, float)` method has **no access modifier** (*package private*) which means, according to Java language specification, that it is only accessible within classes in the same package, therefore that method is not visible by any test set (we are assuming that test code, generally located into `/test` directory, is always included into a different package respect to application code, which is conversely located into `/main` directory)

Therefore, we can conclude by stating that:

$$\textbf{Statement Coverage} = \frac{|S_c|}{|S_e| - |S_i|} = \frac{85}{89 - 4} = 1 = 100\% \quad (4)$$

Since statement coverage of T is 1, we can consider T as **adequate with respect to the statement coverage criterion**.

2.3.2 *Decision coverage (Branch decision coverage).*

According to [6], a *decision* is considered covered if the flow of control has been diverted to all possible destinations that correspond to this decision, i.e. all outcomes of the decision have been taken. This implies that, for example, the expression in the `if` or `while` statement has evaluated to `true` in some execution and to `false` in the same or another execution. Note that each `if` and each `while` contribute to *one* decision whereas a `switch` may contribute to more than one.

According to our analysis, `DiskChecker` class contains 14 possible decisions, therefore $|D_e| = 14$ where D_e is the set of decisions in the program. Our test set T can cover all decision, therefore $|D_c| = 14$, where D_c is the set of decisions covered.

$$\textbf{Decision Coverage} = \frac{|D_c|}{|D_e| - |D_i|} = \frac{14}{14 - 0} = 1 = 100\% \quad (5)$$

Since decision coverage of T is 1, we can consider T as **adequate with respect to the decision coverage criterion**.

---

[8]https://github.com/jacoco/jacoco

Table 6: Equivalence classes and representatives of `DiskChecker` method

Parameter		Equivalence Classes	Representatives
threshold	vEC_1	$0 < \text{warnThreshold} \leq \text{threshold} < 1$	warnThreshold = threshold = 0.5
	iEC_1	threshold ≤ 0	0
	iEC_2	threshold ≥ 1	1
	iEC_3	$0 < \text{threshold} < \text{warnThreshold} < 1$	$0 < \text{warnThreshold} - 0.1 < 1$
warnThreshold	vEC_1:	warnThreshold $\leq$ threshold	threshold
	iEC_1	warnThreshold $>$ threshold	threshold + 0.1

Table 7: Test cases of `DiskChecker` method

Test Case	Parameter		Expected output	Actual output	Passed
	threshold	warnThreshold			
$Valid_1$	0.95	0.95	No Exception	No Exception	✓
$Invalid_1$	0	0.5	Exception	Exception	✓
$Invalid_2$	1	0.5	Exception	Exception	✓
$Invalid_3$	0.5	0.6	Exception	Exception	✓

Table 8: Equivalence classes and representatives of `checkDir` method

Parameter		Equivalence Classes	Representatives
dir	vEC_1	A not-null File object, which: • Represents a valid directory • Exists • BookKeeper's user has read permissions • BookKeeper's user has write permissions	createTempDir("directoryFile", "test")
	vEC_2	A not-null File object, which: • Represents a valid directory • Not exists • BookKeeper's user has write permissions to make that directory	new File("./makeMultiple/dir/path")
	iEC_1	A not-null File object, which: • It does **not** represent a directory, that is it can be one of the following: (1) regular file (2) symbolic link (3) character device file (4) ...	new File("/dev/zero")
	iEC_2	A not-null File object, which: • Represents a valid directory • Not exists • BookKeeper's user has **not** write permissions to make that directory	new File("/root/notMakable")
	iEC_3	A not-null File object, which: • Represents a valid directory • Exists • BookKeeper's user has **not** read permissions	new File("/root")
	iEC_4	A not-null File object, which: • Represents a valid directory • Exists • BookKeeper's user has **not** write permissions	new File("/home")

	iEC_5	A not-null File object, which:
		• Does not represent a valid directory (from file system point of view) because contains forbidden characters like:
		(1) For NTFS file system:
		   \ / : * ? < > |
		(2) For Btrfs, ext4, ext3, XFS file systems:
		   NULL /
			new File("\u0000") (NULL)

	iEC_6	A null object.	null

Table 9: Test cases of `checkDir` method

Test Case	Parameter dir	Expected output	Actual output	Passed
$Valid_1$	`IOUtils.createTempDir("directoryFile", "test")`	No Exception	No Exception	✓
$Valid_2$	`new File("./makeMultiple/dir/path")`	No Exception	No Exception	✓
$Invalid_1$	`new File("/dev/zero")`	Exception	Exception	✓
$Invalid_2$	`new File("/root/notMakable")`	Exception	Exception	✓
$Invalid_3$	`new File("/root")`	Exception	Exception	✓
$Invalid_4$	`new File("/home")`	Exception	Exception	✓
$Invalid_5$	`new File("\u0000")`	Exception	Exception	✓
$Invalid_6$	`null`	Exception	Exception	✓

Table 10: Equivalence classes and representatives of `getTotalDiskUsage` method

Parameter		Equivalence Classes	Representatives

`dirs`	vEC_1	A not-null `List<File>` object, which:
		• `dirs.size() > 0`
		• $\forall x \in dirs$ is true that:
		  – x exists
		  – x represents a valid directory
		  – BookKeeper's user has permissions to read x
			(see the code)

	iEC_1	A not-null `List<File>` object, where: • `dirs.size() = 0`	`new new ArrayList<>()`
	iEC_2	A not-null `List<File>` object, where: • `dirs.size() > 0` • $\exists x \in dirs : x$ is not a directory, that is x can be: (1) regular file (2) symbolic link (3) character device file (4) ...	*(see the code)*
	iEC_3	A not-null `List<File>` object, where: • `dirs.size() > 0` • $\exists x \in dirs : x$ not exist	*(see the code)*
	iEC_4	A not-null `List<File>` object, where: • `dirs.size() > 0` • $\exists x \in dirs : x = $ `null`	*(see the code)*
	iEC_5	A not-null `List<File>` object, where: • `dirs.size() > 0` • $\exists x \in dirs :$ BookKeeper's user has not permissions to read x	*(see the code)*
	iEC_6	A not-null `List<File>` object, where: • `dirs.size() > 0` • $\exists x \in dirs : x$ does not represent a valid directory (from file system point of view).	*(see the code)*
	iEC_7	A null object.	`null`

2.3.3 Condition coverage. Unlike decision coverage, condition coverage ensures that each simple condition within a compound condition has assumed both values `true` and `false`.

Le be given:

C_e the set of simple conditions in the program.
C_c the set of of simple conditions covered by our test set T.
C_i the set infeasible simple conditions.

According to JaCoCo and sonarcloud reports, our test set T covers up to 46 simple conditions, out of a total of 46, while $|C_i| = 0$. Therefore we can say that:

$$\textbf{Condition Coverage} = \frac{|C_c|}{|C_e| - |C_i|} = \frac{46}{46 - 0} = 1 = 100\% \quad (6)$$

Since condition coverage is equal to 1, our test set T is adequate with respect to the condition coverage criterion.

2.4 Mutation Analysis

According to PIT [9] report, mutation coverage equal to 63, con 46

---

[9] https://pitest.org/

Table 11: Test cases of `onClusterChanged` method

Test Case	Parameter		Expected output	Actual output	Passed
	writableBookies	readOnlyBookies			
$Valid_1$	Valid Set<BookieSocketAddress> object (size = 0).	Valid Set<BookieSocketAddress> object (size = 0).	Empty Set<BookieSocketAddress>	Empty Set<BookieSocketAddress>	✓
$Valid_2$	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	Empty Set<BookieSocketAddress>	Empty Set<BookieSocketAddress>	✓
$Valid_3$	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object (knownBookies field of a DefaultEnsemblePlacementPolicy instance have size equal to 5)	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	Not empty Set<BookieSocketAddress> (size = 4)	Not empty Set<BookieSocketAddress> (size = 4)	✓
$Invalid_1$	Valid Set<BookieSocketAddress> object inside which there is one null reference.	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	Exception	Empty Set<BookieSocketAddress>	✗
$Invalid_2$	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	Valid Set<BookieSocketAddress> object inside which there is one null reference.	Exception	Empty Set<BookieSocketAddress>	✗
$Invalid_3$	null	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	Exception	Exception	✓
$Invalid_4$	Valid Set<BookieSocketAddress> object inside which there is 1 valid BookieSocketAddress object.	null	Exception	Exception	✓

REFERENCES

[1] [n.d.]. BookKeeper concepts and architecture. https://bookkeeper.apache.org/docs/4.8.2/getting-started/concepts/. [Online; accessed 4-October-2020].

[2] [n.d.]. BookKeeper configuration. https://bookkeeper.apache.org/docs/4.7.3/reference/config/. [Online; accessed 6-October-2020].

[3] [n.d.]. The BookKeeper protocol. https://bookkeeper.apache.org/docs/4.8.2/development/protocol/. [Online; accessed 5-October-2020].

[4] [n.d.]. Interface EnsemblePlacementPolicy. https://bookkeeper.apache.org/docs/4.8.2/api/javadoc/org/apache/bookkeeper/client/EnsemblePlacementPolicy.html. [Online; accessed 5-October-2020].

[5] Aalok Ahluwalia, Massimiliano Di Penta, and Davide Falessi. 2020. On the Need of Removing Last Releases of Data When Using or Validating Defect Prediction Models.

[6] Andreas Spillner, Tilo Linz, and Hans Schaefer. 2011. *Software Testing Foundations: A Study Guide for the Certified Tester Exam* (3rd ed.). Rocky Nook.

[7] M. van Steen and A.S. Tanenbaum. 2017. *Distributed Systems, 3rd ed.* distributed-systems.net.