Report ISW2 - A.A. 2019-2020

Modulo Software Testing

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 [6].

In particular, for our testing activities involving Apache BookKeeper™, 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.

Regarding Apache OpenJPA™, we have canalised following classes:

`SimpleRegex` [6] Marked as defective and, fortunately, our testing activity has discovered many defects.

`ClassUtil` [7] Not been marked as defective.
We have choose this class due to its relative high LOC value (213) and, primarily, owing to age (214 weeks) which, compared to the average age of all project's classes (661.72 weeks), we believe being very low.
Generally, younger classes belonging to latest releases can be affected by dormant (not discovered) defects[6].
Our testing activity has found very few 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 [9].

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.

[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`
[6] `org.apache.openjpa.lib.util.SimpleRegex`
[7] `org.apache.openjpa.lib.util.ClassUtil`

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 [9]. 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[9]:

(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.

1.2.1 Notation.

Before proceeding with our analysis, it's important to make some clarifications.

For all analysed class, we have developed several test sets, every of which is denoted as T_x with $x = \{1, 2, 3, \ldots\}$.

Moreover, is true that:

$$\forall x, y \in \mathbb{Z}^{+} : y > x \Longrightarrow T_x \subset T_y \quad (1)$$

Above statement states that T_y contains all test cases developed for T_x including some more. Is important to precise that T_y represents an "*improved*" version of T_x because, running T_y, we can achieve better results respect to our adequacy criteria. The aim of that notation is to highlight the improving of our test sets during various step of our testing activity.

In particular, we have used T_1 to denote the initial test set developed during classes equivalence identification step. All improved test sets, denoted as T_2, T_3, T_4 etc., were been developed during both mutation analysis and during the improving of our results related to our adequacy criteria. For each improved test set, we will indicate which method was added or changed.

1.2.2 Existent test set removal.

As request, we have removed all existing tests inside each aforementioned project.

To do that, we have simply exploited following `bash` command, invoked at the root directory of the each project:

```
rm -R $(find ./ -type d -name "test")
```

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 update[10]. 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[10]. 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:

$$E \geqslant Q_w \geqslant Q_a \quad (2)$$

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 *Methods testing analysis.*

2.1.1.1 `initialize`.

According to documentation, after the creation of an `DefaultEnsemblePlacementPolicy` 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**[8] to manage `conf` parameter's return values.

All valid and invalid equivalence classes, indicated as vEC and iEC respectively, are reported in **table 1**. We have developed 2 valid test cases and 1 negative test case. All test passed.

Listing 1: Signature of method `initialize`

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

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

2.1.1.2 `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<BookieSocketAddress>` 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.

2.1.1.3 `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 to documentation, `onClusterChanged` should be invoked

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	$\texttt{ensembleSize} = 0$	0
	vEC_2	$0 < \texttt{ensembleSize} \leq \|K\| - \|E\|$	$\|K\| - \|E\|$
	iEC_1	$\texttt{ensembleSize} < 0$	-1
	iEC_2	$\texttt{ensembleSize} > \|K\| - \|E\|$	$\|K\| - \|E\| + 1$
`writeQuorumSize`	vEC_1	$\texttt{writeQuorumSize} \leq \texttt{ensembleSize}$	`ensembleSize`
	iEC_1	$\texttt{writeQuorumSize} > \texttt{ensembleSize}$	$\texttt{ensembleSize} + 1$
`ackQuorumSize`	vEC_1	$\texttt{ackQuorumSize} \leq \texttt{writeQuorumSize}$	`writeQuorumSize`
	iEC_1	$\texttt{ackQuorumSize} > \texttt{writeQuorumSize}$	$\texttt{ackQuorumSize} + 1$
`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: • $0 < \|E\| < \|K\|$ • $E \subset K$ • $\|K\| - \|E\| \geq \texttt{ensembleSize}$	*(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: • $\|E\| > 0$ • $\exists x \in E : x = \texttt{null}$	*(see the code)*
	iEC_3	`null` object	`null`

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.

2.1.1.4 `updateBookieInfo`.

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

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` $\cap$ `readOnlyBookies` $= \emptyset$	*(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` $\cap$ `readOnlyBookies` $= \emptyset$	*(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`

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)
```

2.1.1.5 `replaceBookie`.

Documentation reports that `replaceBookie` is choose randomly a 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 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`

2.1.1.6 `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.

Oblivious, for purposes related to our adequacy criteria, we have developed one test case for each above method, passing random values as parameters.

2.1.2 Adequacy Criteria.

Is our test set T good enough? Has `DefaultEnsemblePlacementPolicy` class been tested thoroughly?

In general, test adequacy refers to the *goodness* of a test set, which must be measured against a quantitative criterion. As known, a test adequacy criterion could be based either on *requirements* or the *implementation* of the program under test. Particularly, for all test objects analysed in this report, we have adopted several **white-box** test adequacy criteria, that is depend solely on the implementation of our test objects.

2.1.2.1 Statement Coverage.

The statement coverage of a test set T is computed as following[8]:

$$\textbf{Statement Coverage} = \frac{|S_c|}{|S_e| - |S_i|} \quad (3)$$

Where:

S_c is the set of statements covered.
S_i is the set of unreachable statements.
S_e is the set of statements in the program.

Let's start with our test set T_1, that is the test set containing all valid and invalid test cases built during the analysis of the equivalence classes made previously.

According to `sonarcloud` and `JaCoCo` reports, our test set T_1 is able to cover up to 72 lines statements out of a total of 82, giving to us a statement coverage equal to 0.963 (96.3%)

To improve statement coverage, we have add following unit test, obtaining test set T_2:

- `additionalTestCase_1()` (`TestOnClusterChanged` class)

Fortunately, test set T_2 is able to cover up to 81 lines statements out of a total of 82.

However, we are unable to cover all statements because, we believe, in `newEnsemble` method is present an *unreachable* statements, because it falls on an *infeasible path*, that is a path that would never be reached by any test set with any type of input data.

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**	

Therefore, since $|S_i| = 1$, according to **eq. (3)**, statement coverage is equal to 1 (100%).

Since statement coverage of T_2 is 1, we can consider T_2 as **adequate with respect to the statement coverage criterion**.

2.1.2.2 Decision coverage (Branch decision coverage).

According to [8], 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 [8].

Decision coverage of a test set T is computed as following[8]:

$$\textbf{Decision Coverage} = \frac{|D_c|}{|D_e| - |D_i|} \qquad (4)$$

Where:

D_c is the set of decisions covered.
D_i is the set of unreachable decisions.
D_e is the set of decisions in the program.

According to our analysis, our test set T_2 is able to cover 14 decisions out of a total of 15. In `newEnsemble` method, there is an unreachable decisions, therefore, since $|D_i| = 1$, our test set T_2 give to us a decision coverage equal to 1 (100%).

Since decision coverage of T_2 is 1, we can consider T_2 as **adequate with respect to the decision coverage criterion**.

2.1.2.3 Condition coverage.
Unlike decision coverage, condition coverage ensures that each simple condition within a *compound* condition has assumed both values `true` and `false`[8].

Condition coverage of a test set T is computed as following[8]:

$$\textbf{Condition Coverage} = \frac{|C_c|}{|C_e| - |C_i|} \tag{5}$$

Where:

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.

Our test set T_2 is able to cover 33 decisions out of a total of 34. Unfortunately, in `newEnsemble` method there is an infeasible conditions (a simple condition inside a `while` statement which is never `false`), therefore, since $|C_i| = 1$, our test set T_2 give to us a condition coverage equal to 1 (100%).

Since condition coverage of T_2 is 1, we can consider T_2 as **adequate with respect to the condition coverage criterion**.

2.1.3 Mutation Analysis.
So, mutation testing represents a very powerful technique to achieve two goals[7]:

(1) Evaluating test suite quality.
(2) Once we have a good test suite, executing its test cases against the original program to find errors.

In mutation analysis, from a program p, a set of faulty programs p_m, called *mutants*, is generated by a few single syntactic changes to the original program p. Assume that a test set T is supplied to the system. In this case each, mutant p_m will then be run against this test set T. If the result of running p_m is different from the result of running p for any test case in T, then the mutant p_m is said to be *killed*, otherwise, it is said to have *survived*[7].

Our objective, is to improve our test set T providing additional test inputs (or improving existing ones) to kill these surviving mutants.

As known, to evaluate the goodness of a test suite, an adequacy criterion, known as the *Mutation Score* (or *Mutation Coverage*), is used.

So, according to `PIT` [9] report, mutation coverage of our T_2 test set is equal to 28, out of a total of 41 (mutation score 68.29%). This result means that, after that each mutant is been run against our test set T_2, some mutants *survived*.

[9]https://pitest.org/

Therefore, to improving our test suite, we have add following unit tests to our test set T_2:

- `additionalTestCase_1()` (`TestInitialize` class)
- `additionalTestCase_1()` (`TestOther` class)

Moreover, we have improved following unit test:

- `test_1` (`TestOther` class)
- `test_3` (`TestOther` class)
- `test_4` (`TestOther` class)

After these changes, with our new test set T_3, we have killed 4 extra mutants, obtaining a mutation coverage equal to 32, out of a total of 41 (mutation score 78.04%).

As known, the goal of mutation analysis is to raise the mutation score to 1, indicating that a test set T is sufficient to detect all the faults denoted by the mutants[7]. However, this aim is out the scope of our project.

2.1.4 Conclusions. Finally, using our tests set T_3, 23 test, out of a total of 86, failed, revealing the presence of several bugs, confirming results made by defect prediction model.

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:

$$\begin{aligned} 0 <& \quad D \quad < 1 \\ D_w \leq& \quad D \end{aligned} \tag{6}$$

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

2.2.1 Methods testing analysis.

2.2.1.1 `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**.

From identified equivalence classes, we have developed 1 valid test case and 3 negative test cases. All tests passed. Test cases are reported in **table 7**.

Listing 6: Signature of method `DiskChecker`

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

2.2.1.2 `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:

$$\texttt{usage} = 1 - \frac{\texttt{usableSpace}}{\texttt{totalSpace}} \tag{7}$$

All valid and invalid equivalence classes are reported in table **table 8**, while method signature is reported in **listing 7**. We have developed 2 valid test cases and 5 negative test cases, with a total 8 test cases, all passed. Test cases are described in **table 9**.

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

Listing 7: Signature of method `checkDir`

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

2.2.1.3 `getTotalDiskUsage`.
`getTotalDiskUsage` is used to compute disk usage fraction, already defined in **7**, 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**. We have developed 1 valid test case and 7 negative test cases, with a total of 8 test cases, of which 2 failed.

Listing 8: Signature of method `getTotalDiskUsage`

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

2.2.1.4 `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, methods comments 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` while, for `getTotalFreeSpace` method, the 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**.

To test `getTotalDiskSpace` method, we have developed 2 valid test cases and 6 negative test cases, with a total 8 test cases, of which 2 failed. Likewise, we have developed the same number of tests for `getTotalFreeSpace` method too; however 3 unit tests failed out of a total of 8

2.2.2 Adequacy Criteria.

2.2.2.1 Statement coverage.

Our test set T_1, made up of all test cases built during equivalence class analysis, is able to cover up to 69 statements out of a total of 89, reaching a statement coverage equal to **0,775** (**77,5%**).

To improve our test set, we add following unit tests to test set T_1:

(1) `additionalTest_1` (`TestDiskChecker` class).
(2) `additionalTest_1` (`TestCheckDir` class).
(3) `additionalTest_2` (`TestCheckDir` class).
(4) `additionalTest_3` (`TestCheckDir` class).

With this new test set, T_2, we are able to cover up to 85 statements out of a total of 89.

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

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

Table 7: Test cases of `DiskChecker` method

Test Case	Parameter `threshold`	Parameter `warnThreshold`	Expected output	Actual output	Passed
Valid$_1$	0.5	0.5	No Exception	No Exception	✓
Invalid$_1$	0	0	Exception	Exception	✓
Invalid$_2$	1	1	Exception	Exception	✓
Invalid$_3$	0.5	0.6	Exception	Exception	✓

However, `DiskChecker` class contains a method, called `setDiskSpaceThreshold(float, float)`, which, we believe, 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, in this case, 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 our test set.

Therefore, we can conclude by stating that $|S_i| = 4$ and our test set T_2 gives to us a statement coverage equal to **1** (**100%**).

Since statement coverage of T_2 is 1, we can consider T_2 as **adequate with respect to the statement coverage criterion**.

2.2.2.2 Decision coverage (Branch decision coverage).

According to our analysis, `DiskChecker` class contains 16 decision. Since our test set T_2 covers all decisions, decision coverage is equal to **1** (**100%**). Because decision coverage of T_2 is 1, we can consider T_2 as **adequate with respect to the decision coverage criterion**.

2.2.2.3 Condition coverage.

According to sonarcloud reports, our test set T_2 covers up to 45 simple conditions, out of a total of 46 with a condition coverage equal to 0.978 ($97, 8\%$).

Since condition coverage is less than 1, our test set T_2 is *not* adequate with respect to the condition coverage criterion.

2.2.3 Mutation Analysis.

According to `PIT` report, mutation coverage of our T_2 test set is equal to 29, out of a total of 46 (mutation score 63.04%).

To build our new test set T_3, we added following unit tests to T_2:

- `additionalTest_1` (`TestTest` class)
- `additionalTest_2` (`TestTest` class)
- `additionalTest_3` (`TestTest` class)

After these changes, with our new test set T_3, we have killed 4 extra mutants, obtaining a mutation coverage equal to 34, out of a total of 41 (mutation score 82.92%).

2.2.4 Conclusions. Finally, using our test set T_3, 7 tests, out of a total of 50, failed revealing the presence of several bugs, despite its stability according to our prediction model.

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

<table>
<tr><th>Parameter</th><th colspan="2">Equivalence Classes</th><th>Representatives</th></tr>
<tr><td rowspan="7">dir</td><td>vEC_1</td><td>A not-null File object, which:
• Represents a valid directory
• Exists
• BookKeeper’s user has read permissions
• BookKeeper’s user has write permissions</td><td>createTempDir("directoryFile", "test")</td></tr>
<tr><td>vEC_2</td><td>A not-null File object, which:
• Represents a valid directory
• Not exists
• BookKeeper’s user has write permissions to make that directory</td><td>new File("./makeMultiple/dir/path")</td></tr>
<tr><td>iEC_1</td><td>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) ...</td><td>new File("/dev/zero")</td></tr>
<tr><td>iEC_2</td><td>A not-null File object, which:
• Represents a valid directory
• Not exists
• BookKeeper’s user has not write permissions to make that directory</td><td>new File("/root/notMakable")</td></tr>
<tr><td>iEC_3</td><td>A not-null File object, which:
• Represents a valid directory
• Exists
• BookKeeper’s user has not read permissions</td><td>new File("/root")</td></tr>
<tr><td>iEC_4</td><td>A not-null File object, which:
• Represents a valid directory
• Exists
• BookKeeper’s user has not write permissions</td><td>new File("/home")</td></tr>
<tr><td>iEC_5</td><td>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 /</td><td>new File("\u0000") (NULL)</td></tr>
<tr><td></td><td>iEC_6</td><td>A null object.</td><td>null</td></tr>
</table>

3 APACHE OPENJPA™

3.1 `SimpleRegex` class testing analysis

As the name suggests, `SimpleRegex` class is used to manage a ”*simple*” RegEx, that is a ***Reg****ular* ***Ex****pression*, which represents a sequence of characters that defines a search pattern used in several string-searching algorithms or during string-input validation operations; in other words, the aim of a RegEx pattern is to match a target string.

An example of RegEx is `gr(a|e)y` which matches both `gray` or `grey` strings.

A RegEx pattern is made up of a sequence of elements including:

Atoms which represents a regular character that has a literal meaning.

Meta-characters which represents an element with a very special meaning, depending on context. The majority of RegEx processor engines supports up to 14 meta-characters including:

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`

`$ & / ( ) [ ] ^ @ \ | + - *`

According to documentation, the RegEx processor engine of the `SimpleRegex` class supports only 2 meta-characters:

- `.` which matches any single character.
  This meta-characters is known as *wildcard*. For example, `a.b` matches any string that contains an "a", then any other character and then "b", like `axb`.
- `.*` which matches any string.
  For example, `a.*b` matches any string that contains an "a", and then the character "b" at some later point, like `a andrea graziani b`.

Made this necessary introduction, let's start building our test set T_1, analysing all public methods.

3.1.1 Methods testing analysis.

3.1.1.1 `SimpleRegex`.

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

- **`expr`** A `string` object containing the RegEx.
- **`caseInsensitive`** To enable or disable case sensitive.

All valid and invalid equivalence classes are reported in **table 11**, while method signature is shown below, in **listing 9**.

From identified equivalence classes, we have developed $1 \times 2 = 2$ valid test cases and $1+1+1+1+1 = 5$ negative test cases. Precisely, we have tested each negative test cases when `caseInsensitive` parameter is both `true` and `false`; therefore, we have built a total of 10 negative test cases.

Out of a total of 12 tests, 7 of them failed: a very bad result.

Listing 9: Signature of method `SimpleRegex`

```
public SimpleRegex(String expr, boolean caseInsensitive)
```

3.1.1.2 `matches`.

This method is used to check the matching between RegEx expression and `target` string.

All valid and invalid equivalence classes are reported in **table 12**, while method signature is shown in **listing 10**. We have developed 1 valid test case and 1 negative test case, both run twice using different values of `caseInsensitive` parameter described previously. All tests passed.

Listing 10: Signature of method `matches`

```
public boolean matches(String target)
```

3.1.2 Adequacy Criteria.

3.1.2.1 Statement coverage.

Our test set T_1 is able to cover up to 43 statements out of a total of 48, reaching a statement coverage equal to **0,895** (**89,5%**).

To improve our test set, we add following unit tests to T_1, obtaining our new test set T_2:

(1) `additionalTest_1` (into `TestMatches` class).
(2) `additionalTest_2` (into `TestMatches` class).
(3) `additionalTest_3` (into `TestMatches` class).

Using T_2, we are able to cover up to 47 statements out of a total of 48, reaching a statement coverage equal to **0,979** (**97,9%**).

Since during our test activities we were unable to build a test set capable to cover last statement (probably *unreachable*), we can state that T_2 in **not** adequate with respect to the statement coverage criterion.

3.1.2.2 Decision coverage (Branch decision coverage).

According to our analysis, `SimpleRegex` class contains 13 decisions and our test set T_2 is able to cover up to 12 of them, reaching a decision coverage equal to **0,923** (**92,3%**).

Because decision coverage of T_2 is less than 1, T_2 is **not** adequate with respect to the decision coverage criterion.

3.1.2.3 Condition coverage.

According to sonarcloud reports, our test set T_2 covers up to 30 simple conditions out of a total of 32, with a condition coverage equal to 0.938 $(93,8\%)$.

Since condition coverage is less than 1, our test set T_2 is **not** adequate with respect to the condition coverage criterion.

3.1.3 Mutation Analysis.

Unfortunately, it's impossible to start a mutation analysis because our testing activity has revealed many errors into original program P. As known, we need a test set which must be *successfully* executed against the original program *before* staring mutation analysis [7]. Nevertheless, according to `PIT` report, our test set T_2 is able to kill 32 mutants, out of a total of 41, achieving a mutation score equal to 78.04%.

Table 11: Equivalence classes and representatives of `SimpleRegex` method

Parameter	Equivalence Classes		Representatives

expr	vEC_1	A not-`null` `String` object which:
		• Is *not* blank, that is `expr.length()` > 0 [1]
		• It may contain alphanumeric characters
		• It may contain following meta-characters *only*: [2]
		`.`
		`.*`
		• It must not contain any *escape character* [3]
			`a.*b`

	iEC_1	A not-`null` `String` object which: • `expr.length()` $= 0$	""
	iEC_2	A not-`null` `String` object which: • `expr.length()` > 0 • It represents an invalid pattern because it contains both unsupported meta-characters and escape characters.	`a.*(r|t)b\r`
	iEC_3	A not-`null` `String` object which: • `expr.length()` > 0 • It represents an invalid pattern because it contains unsupported meta-characters and no escape characters.	`a*`
	iEC_4	A not-`null` `String` object which: • `expr.length()` > 0 • It represents an invalid pattern because it contains escape characters and no unsupported meta-characters	`\t`
	iEC_5	`null` object.	`null`
caseInsensitive	vEC_1	`true`	`true`
	vEC_2	`false`	`false`

[1] According to RegEx specification, an empty pattern is *invalid*, because it either returns empty matches or matches with any string depending on RegEx processor engine. In other words, it has an undefined behaviour. In order to match empty strings, `(^?![\s\S])` is used.
[2] Any other meta-character is considered as invalid because, according to `SimpleRegex` documentation, they are unsupported.
[3] An escape character is a character that invokes an alternative interpretation on the following characters in a character sequence. It is a particular case of meta-characters. For example `\u00A9` is an escape character for ©. `SimpleRegex` class doesn't support them.

Table 12: Equivalence classes and representatives of `matches` method

Parameter	Equivalence Classes		Representatives
`target`	vEC_1	A not-`null` `String` object which • `expr.length()` ≥ 0 • Can contain any character.	`"Andrea1234\t= "`
	iEC_1	`null` object.	`null`

Obliviously, like in previous cases, we have tried to improve our test set adding new unit tests or improving existing ones but, unfortunately, mutants were still surviving. We believe that they are *equivalent mutants* which we cannot kill because they always produce the same output as the original program.

3.1.4 Conclusions.

At the end of our testing activity, using test set T_2, we reports 9 failed unit tests, out of a total of 22. This result, confirming our defect prediction, reveals the presence of several bugs.

3.2 `ClassUtil` class testing analysis

`ClassUtil` represents an utility class used to manage classes. Particularly, that class is used to dynamically loads Java classes, to retrieve classes names and packages names.

How is possible to load dynamically Java classes? Using `ClassLoader` objects!

`ClassLoader` objects are a part of the *Java Runtime Environment* that dynamically loads Java classes into the *Java Virtual Machine*. When a class is requested, the class loader tries to locate the class and load the class definition into the runtime using a fully qualified class name [5].

`ClassLoader` instances use a so-called *delegation approach*: each instance of `ClassLoader` has an associated parent class loader. When requested to find a class or resource, a `ClassLoader` instance will delegate the search for the class or resource to its parent class loader before attempting to find the class or resource itself. A special class loader, called *bootstrap*, is used as a parent of all the other `ClassLoader` instances [5].

Normally, the Java virtual machine loads classes from the local file system in a platform-dependent manner.

For example, on UNIX systems, the virtual machine loads classes from the directory defined by the `CLASSPATH` environment variable.

Made this necessary introduction, let's start building our test set T_1, analysing all public methods.

3.2.1 *Methods testing analysis.*

To be precise, this class contains all `static` methods; therefore no constructor was been provided.

3.2.1.1 `toClass`.

Inside `ClassUtil` class, there are two *overloaded* methods called `toClass`, which signatures are reported in **listing 11** and **listing 12**.

Listing 11: Signature of method `toClass` #1

```
public static Class toClass(String str, ClassLoader loader)
```

Listing 12: Signature of method `toClass` #2

```
public static Class toClass(String str, boolean resolve,
                            ClassLoader loader)
```

We will analyse only the second one, since the first one call the second method with `resolve` parameter equal to `false`.

All valid and invalid equivalence classes are reported in **table 13**.

From identified equivalence classes, we have developed $3 \times 2 \times 2 = 12$ valid test cases and 20 negative test cases, testing all possible combinations. In total, we built 32 tests: all passed.

3.2.1.2 `getPackageName`.

Like previously, there are two *overloaded* methods called `getPackageName`, which signatures are reported in **listing 13** and **listing 14**.

Listing 13: Signature of method `getPackageName` #1

```
public static String getPackageName(Class cls)
```

Listing 14: Signature of method `getPackageName` #2

```
public static String getPackageName(String fullName)
```

All valid and invalid equivalence classes of the first method are reported in **table 15**, while those of the second are reported in **table 14**.

3.2.1.3 `getClassName`.

Like previously, there are two *overloaded* methods called `getClassName`, which signatures are reported in **listing 15** and **listing 16**.

Listing 15: Signature of method `getClassName` #1

```
public static String getClassName(Class cls)
```

Listing 16: Signature of method `getClassName` #2

```
public static String getClassName(String fullName)
```

All valid and invalid equivalence classes of the first method are reported in **table 15**, while those of the second are reported in **table 14**.

3.2.2 *Adequacy Criteria.*

3.2.2.1 Statement coverage.

Our test set T_1 is able to cover up to 52 statements out of a total of 75, reaching a statement coverage equal to **0,693** (**69,3%**).

Since we have obtained a very bad result, in order to improve our test set, we add following unit tests to T_1, obtaining our new test set T_2:

(1) `additionalTest_1` (into `OtherTests` class).
(2) `additionalTest_2` (into `OtherTests` class).
(3) `additionalTest_3` (into `OtherTests` class).
(4) `additionalTest_4` (into `OtherTests` class).
(5) `additionalTest_5` (into `OtherTests` class).
(6) `additionalTest_6` (into `OtherTests` class).

Using T_2, we are able to cover up to 75 statements, reaching a statement coverage equal to **1** (**100%**).

Since statement coverage of T_2 is 1, we can consider T_2 as **adequate with respect to the statement coverage criterion**.

Table 13: Equivalence classes and representatives of `toClass` method

<table>
<tr><th>Parameter</th><th colspan="2">Equivalence Classes</th><th>Representatives</th></tr>
<tr><td rowspan="6">str</td><td>vEC_1</td><td>A not-<code>null</code> <code>String</code> object which:
Is not blank, that is <code>str.length()</code> > 0
Must be a binary name as defined by The Java™Language Specification.[1]
Specified binary name must matches an existent and readable <code>.class</code> (or <code>.jar</code>) file.
In particular:
Into specified <code>str</code> is represent a dollar signs ('<code>$</code>') used to separate inner and outer classes</td><td><code>"javax.swing.JSpinner$DefaultEditor"</code></td></tr>
<tr><td>vEC_2</td><td>A not-<code>null</code> <code>String</code> object which:
<code>str.length()</code> > 0
Must be a binary name as defined by The Java™Language Specification.
Specified binary name must matches an existent and readable <code>.class</code> (or <code>.jar</code>) file.
In particular:
It is a valid binary name containing only periods ('<code>.</code>') used to identify the package name.</td><td><code>"java.lang.String"</code></td></tr>
<tr><td>vEC_3</td><td>A not-<code>null</code> <code>String</code> object which:
<code>str.length()</code> > 0
Must be a binary name as defined by The Java™Language Specification.
Specified binary name must matches an existent and readable <code>.class</code> (or <code>.jar</code>) file.
In particular:
Specified <code>str</code> represents an array classes with one or more brackets ('<code>[</code>' and '<code>]</code>')</td><td><code>"java.awt.Point[][]"</code></td></tr>
<tr><td>iEC_1</td><td>A not-<code>null</code> <code>String</code> object which:
<code>str.length()</code> $= 0$</td><td>""</td></tr>
<tr><td>iEC_2</td><td>A not-<code>null</code> <code>String</code> object which:
<code>str.length()</code> > 0
It represents an invalid binary name.</td><td><code>"$%&andrea..invalid..binary*-+"</code></td></tr>
<tr><td>iEC_3</td><td><code>null</code> object.</td><td><code>null</code></td></tr>
<tr><td rowspan="2">resolve</td><td>vEC_1</td><td><code>true</code>[2]</td><td><code>true</code></td></tr>
<tr><td>vEC_2</td><td><code>false</code></td><td><code>false</code></td></tr>
<tr><td rowspan="3">loader</td><td>vEC_1</td><td>A not-<code>null</code> <code>ClassLoader</code> object which:
Must be able to load specified <code>.class</code> file; that is, loader's class paths must include a directory containing specified <code>str</code> class file.</td><td>(see the code)</td></tr>
<tr><td>vEC_2</td><td><code>null</code> object.[3]</td><td><code>null</code></td></tr>
<tr><td>iEC_1</td><td>A not-<code>null</code> <code>ClassLoader</code> object which:
Is unable to load specified <code>.class</code> class file.</td><td>(see the code)</td></tr>
</table>

[1] According to the *The Java™Language Specification*, any class or interface must be named by its *binary name*, which must meet several constraints. Examples of valid class names include:
- `java.lang.String`
- `javax.swing.JSpinner$DefaultEditor`
- `java.security.KeyStore$Builder$FileBuilder$1`

For details, please visit: https://docs.oracle.com/javase/specs/jls/se8/html/jls-13.html#jls-13.1

[2] When `true`, the specified class will be *initialized* if it has not been initialized earlier. Initialization of a class consists of executing its static initializers and the initializers for static fields declared in the class.
For details, please visit: https://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html

[3] Specified `ClassLoader` object can be `null` because, in that case, bootstrap class loader would be called.

Table 14: Equivalence classes and representatives of `getClassName(String)` and `getPackageName(String)` methods

Parameter		Equivalence Classes	Representatives
fullName	vEC_1	A not-`null` `String` object which: • Is *not* blank, that is `str.length()` > 0 • Must be a *binary name* as defined by *The Java™Language Specification*. In particular: – Into specified `str` is represent a dollar signs ('`$`') used to separate inner and outer classes	`"javax.swing.JSpinner$DefaultEditor"`
	vEC_2	A not-`null` `String` object which: • `str.length()` > 0 • Must be a *binary name* as defined by *The Java™Language Specification*. In particular: – It is a valid *binary name* containing only periods ('`.`') used to identify the package name.	`"java.lang.String"`
	vEC_3	A not-`null` `String` object which: • `str.length()` > 0 • Must be a *binary name* as defined by *The Java™Language Specification*. In particular: – Specified `str` represents an array classes with one or more brackets ('`[`' and '`]`')	`"java.awt.Point[][]"`
	iEC_1	A not-`null` `String` object which: • `str.length()` $= 0$	""
	iEC_2	A not-`null` `String` object which: • `str.length()` > 0 • It represents an invalid *binary name*.	`"$%&andrea..invalid..binary*-+"`
	iEC_3	`null` object.	`null`

Table 15: Equivalence classes and representatives of `getClassName(Class)` and `getPackageName(Class)` methods

Parameter		Equivalence Classes	Representatives
cls	vEC_1	A not-`null` `Class` object.	`"andrea.Test"`
	iEC_1	`null` object.	`null`

3.2.2.2 Decision coverage (Branch decision coverage).

According to our analysis, `SimpleRegex` class contains 29 decisions and our test set T_2 is able to cover up to 28 of them, reaching a decision coverage equal to **0,965** (**96,5%**).

Because decision coverage of T_2 is less than 1, T_2 is **not** adequate with respect to the decision coverage criterion.

3.2.2.3 Condition coverage.

According to sonarcloud reports, our test set T_2 covers up to 59 simple conditions out of a total of 60, with a condition coverage equal to 0.983 $(98,3\%)$.

Since condition coverage is less than 1, our test set T_2 is **not** adequate with respect to the condition coverage criterion.

3.2.3 Mutation Analysis.

According to `PIT` report, mutation coverage of our T_2 test set is equal to 29, out of a total of 71 (mutation score $40,84\%$).

4 LINKS

https://github.com/AndreaG93/ISW2-Reports
https://github.com/AndreaG93/ISW2-bookkeeper
https://github.com/AndreaG93/ISW2-openjpa
https://travis-ci.org/github/AndreaG93/ISW2-bookkeeper
https://travis-ci.org/github/AndreaG93/ISW2-openjpa
https://sonarcloud.io/dashboard?id=AndreaG93_ISW2-bookkeeper
https://sonarcloud.io/dashboard?id=AndreaG93_ISW2-openjpa

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] [n.d.]. `ClassLoader` documentation. https://docs.oracle.com/javase/7/docs/api/java/lang/ClassLoader.html. [Online; accessed 15-October-2020].

[6] 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.

[7] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[8] A.P. Mathur. 2013. *Foundations of Software Testing, 2/e*. Pearson Education India. https://books.google.it/books?id=OUA8BAAAQBAJ

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

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