IWS2 Projects A.A. 2019-2020

Andrea Graziani
Università degli Studi di Roma "Tor Vergata"
Laurea Magistrale in Ingegneria Informatica
Rome, Italy
andrea.graziani93@outlook.it

Index Terms

component, formatting, style, styling, insert

I. ORG.APACHE.BOOKKEEPER.UTIL.DISKCHECKER ANALYSIS

www.google.it

According to our defect prediction model, `DiskChecker`[1] class was reported as a *buggy* therefore, since it is likely that it exhibits a defect, we have focused on aforementioned class in order to check if there are any errors in it, attempting to increase the *reliability* of the class, that is the probability of failure free execution of that class under given conditions.

Our testing activity has confirmed our defect prediction because, using our test set T_1 which is retrievable from commit `bdf2fc28d2`[2], was reported that up to 5 unit tests have failed, out of a total of 31, revealing the presence of several bugs. In following subsection we will describe how we have build test set T_1.

A. Equivalence Class Partitioning

In order to build our test set T_1, we have adopted a *black box* 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 [1].

Although equivalence class definition is based on specifications (of requirements) only [1], we have defined our test looking the code too, due to the lack of specification about `DiskChecker` class.

1) `DiskChecker(float, float)`: Obliviously, that method represent the constructor of `DiskChecker` class, which manage directories belonging to ledger entities.

According to bookkeeper specification[3], is possible to specify, for each ledger directory, a maximum disk space (`diskUsageThreshold`) which can be used. Moreover is possible to set a warning threshold for disk usage (`diskUsageWarnThreshold`). Specification establishes

that valid values should be in between 0 *and* 1 *(exclusive).*

For both thresholds, bookkeeper specifications establish a default value, which is equal to $0,95$. In table I all found equivalence classes, both for valid (vEC) and invalid input (iEC) , are been reported, including respective representative values necessary to build our test set.

The next step is to combine the values to test cases. From a methodological point of view, 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*).

Generally, since even a few parameters can generate hundreds of valid test cases, o reduce the number of tests, we have adopted following rules

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.

Using the previously given rules, we get $1 \times 1 = 1$ valid test cases (by combining the representatives of the valid equivalence classes) and $1+3=4$ negative test cases (by separately testing representatives of every invalid class). 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.

TABLE I
EQUIVALENCE CLASSES AND REPRESENTATIVES OF DISKCHECKER METHOD

Parameter	Equivalence Classes		Representative
threshold	vEC_1:	$0 <$ warnThreshold $\leq x < 1$	0.5
	iEC_1:	$x \leq 0$	0
	iEC_2:	$x \geq 1$	1
	iEC_3:	$0 < x <$ warnThreshold < 1	0.5 (while warnThreshold is 0.6)
warnThreshold	vEC_1:	$x \leq$ threshold	0.5 (while threshold is 0.5)
	iEC_1:	$x >$ threshold	0.6 (while threshold is 0.5)

TABLE II
TEST CASES OF DISKCHECKER METHOD

Test Case	Parameter threshold	Parameter warnThreshold	Expected output	Actual output	Passed
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 III
EQUIVALENCE CLASSES AND REPRESENTATIVES OF CHECKDIR METHOD

Parameter	Equivalence Classes		Representatives
dir	vEC_1:	Valid File object representing a valid directory (existent, readable and writeable).	IOUtils.createTempDir("directoryFile", "test")
	vEC_2:	Valid File object representing a not existent, but makable, directory.	new File("./makeMultiple/dir/path")
	iEC_1:	Valid File object which does not represent a directory, that is it can be a regular file, a symbolic link, a character device file etc.	new File("/dev/zero")
	iEC_2:	Valid File object representing a non-existent and non-makable directory.	new File("/root/notMakable")
	iEC_3:	Valid File object representing an existent, not readable directory.	new File("/root")
	iEC_4:	Valid File object representing an existent, not writeable directory.	new File("/home")
	iEC_5:	Valid File object representing an invalid directory (from file system point of view because, for example, it contains forbidden characters).	new File("\u0000")
	iEC_6:	A null object.	null

TABLE IV
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 V
EQUIVALENCE CLASSES AND REPRESENTATIVES OF GETTOTALFREESPACE AND GETTOTALDISKSPACE METHODS

Parameter	Equivalence Classes		Representatives
dir	vEC_1:	A valid non-empty List<File> object containing File objects every of which represents a valid (existent and readable) directory.	*See the code*
	vEC_2:	A valid empty List<File> object.	new ArrayList<>()
	iEC_1:	A valid non-empty List<File> object in which at least one File object does not represent a directory, that is it can represent regular file, symbolic link, character device file etc.	*See the code*
	iEC_2:	A valid non-empty List<File> object in which at least one File object represent a not existent directory.	*See the code*
	iEC_3:	A valid non-empty List<File> object in which at least one File object represent an existent but not readable directory.	*See the code*
	iEC_3:	A valid non-empty List<File> object in which at least one File object represent an invalid directory (from file system point of view).	*See the code*
	iEC_3:	A valid non-empty List<File> object in which at least one File object is null.	*See the code*
	iEC_5:	A null object.	null

TABLE VI
EQUIVALENCE CLASSES AND REPRESENTATIVES OF GETTOTALDISKUSAGE, GETTOTALFREESPACE, GETTOTALDISKSPACE METHODS

Parameter	Equivalence Classes		Representatives
dir	vEC_1:	A valid non-empty List<File> object containing File objects every of which represents a valid (existent, readable and writeable) directory.	-
	iEC_1:	A validgfdg non-empty List<File> object containing File objects every of which represents a not existent directory.	-
	iEC_2:	A valid non-empty List<File> object containing File objects every of which represents an existent, not readable and not writeable directory.	-
	iEC_3:	A valid non-empty List<File> object containing File objects every of which not represents a directory, that is it can represent regular file, symbolic link, character device file etc.	-
	iEC_4:	A valid empty List<File> object.	new ArrayList<>()
	iEC_5:	A null object.	null

2) *checkDir(File dir):*
3) *getTotalDiskSpace(List<File> dirs):* JaCoCo [4]

B. Adequacy Criteria

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:

[1] org.apache.bookkeeper.util.DiskChecker
[2] https://github.com/AndreaG93/ISW2-bookkeeper/tree/bdf2fc28d216b0adf9e467cfb51c4f334e84d5c9
[3] https://bookkeeper.apache.org/docs/4.7.3/reference/config/
[4] https://github.com/jacoco/jacoco

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

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

2) Decision coverage (Branch decision coverage): According to [1], 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.

$$\textbf{Decision Coverage} = \frac{|D_c|}{|D_e| - |D_i|} = \frac{14}{14 - 0} = 1 = 100\% \tag{2}$$

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

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:

$$\textbf{Condition Coverage} = \frac{|C_c|}{|C_e| - |C_i|} = \frac{46}{46 - 0} = 1 = 100\% \tag{3}$$

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

C. Mutation Analysis

According to `PIT` [5] report, mutation coverage equal to 63, con 46

II. `ORG.APACHE.BOOKKEEPER.CLIENT.DEFAULTENSEMBLEPLACEMENTPOLICY` ANALYSIS

What is meant by `DefaultEnsemblePlacementPolicy`? What are its responsibilities?

According to `bookkeeper` specifications [2], 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 [3]. 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 [3]. 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 [4], the following invariant must hold:

$$E \geqslant Q_w \geqslant Q_a \tag{4}$$

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.

How are ensemble built in order to be compliant to above specifications?

Is very important to precise that `bookkeeper` uses several algorithms to selects a number of bookies from a cluster as an ensemble, some of which are capable to exploit several network topology proprieties too. According to `bookkeeper` design, implementations of these algorithm must be compliant to `EnsemblePlacementPolicy` interface [5], in other words any implementation must respect the *contract* establish by `EnsemblePlacementPolicy` interface, which covers aspects related to initialization and bookie selection for data placement and reads [5].

[5] https://pitest.org/

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

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

The subject of our analysis is the `DefaultEnsemblePlacementPolicy` class, which encapsulates the simplest algorithm for bookie selection for ensemble creation because it, simply, picks bookies randomly in order to build an ensemble.

According to our defect prediction model, `DefaultEnsemblePlacementPolicy`[6] class was reported as a *buggy* therefore, since it is likely that it exhibits a defect, we have focused on aforementioned class in order to check if there are any errors in it, attempting to increase the *reliability* of the class, that is the probability of failure free execution of that class under given conditions.

A. Equivalence Class Partitioning

1) `onClusterChanged`:

Listing 1. `onClusterChanged` method signature.

```
Set<BookieSocketAddress> onClusterChanged(Set<BookieSocketAddress> writableBookies, Set<BookieSocketAddre
```

TABLE VII
EQUIVALENCE CLASSES AND REPRESENTATIVES OF `ONCLUSTERCHANGED` METHOD

Parameter	Equivalence Classes		Representatives
`writableBookies`	vEC_1:	Valid `Set<BookieSocketAddress>` object (size $\geqslant$ 0) containing valid `BookieSocketAddress` objects.	*(see the code)*
	iEC_1:	Valid `Set<BookieSocketAddress>` object (size $\geqslant$ 0) containing `null` items or invalid `BookieSocketAddress` objects.	*(see the code)*
	iEC_2:	`null` object	`null`
`readOnlyBookies`	vEC_1:	Valid `Set<BookieSocketAddress>` object (size $\geqslant$ 0) containing valid `BookieSocketAddress` objects.	*(see the code)*
	iEC_1:	Valid `Set<BookieSocketAddress>` object (size $\geqslant$ 0) containing `null` items or invalid `BookieSocketAddress` objects.	*(see the code)*
	iEC_2:	`null` object	`null`

2) `newEnsemble`: The @NotNull Annotation is, actually, an explicit contract declaring the following:

A method should not return null. A variable (like fields, local variables, and parameters) cannot should not hold null value.

3) `replaceBookie`:

4) `updateBookieInfo`: According to `DefaultEnsemblePlacementPolicy` interface's documentation [5], `updateBookieI` is used to update bookie info details taking only one input parameter, a `Map<BookieSocketAddress, BookieInfo>` object. Method signature is the following:

Listing 2. `onClusterChanged` method signature.

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

REFERENCES

[1] A. Spillner, T. Linz, and H. Schaefer, *Software Testing Foundations: A Study Guide for the Certified Tester Exam*, 3rd ed. Rocky Nook, 2011.
[2] "Bookkeeper concepts and architecture," https://bookkeeper.apache.org/docs/4.8.2/getting-started/concepts/, [Online; accessed 4-October-2020].
[3] M. van Steen and A. Tanenbaum, *Distributed Systems, 3rd ed.* distributed-systems.net, 2017.
[4] "The bookkeeper protocol," https://bookkeeper.apache.org/docs/4.8.2/development/protocol/, [Online; accessed 5-October-2020].
[5] "Interface `EnsemblePlacementPolicy`," https://bookkeeper.apache.org/docs/4.8.2/api/javadoc/org/apache/bookkeeper/client/EnsemblePlacementPolicy.html, [Online; accessed 5-October-2020].

[6] `org.apache.bookkeeper.client.DefaultEnsemblePlacementPolicy`

TABLE VIII
TEST CASES OF ONCLUSTERCHANGED METHOD

Test Case	Parameter: writableBookies	Parameter: readOnlyBookies	Expected output	Actual output	Passed
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	✓

TABLE IX
EQUIVALENCE CLASSES AND REPRESENTATIVES OF UPDATEBOOKIEINFO METHOD

Test Case	Parameter: writableBookies	Parameter: readOnlyBookies	Expected output	Actual output	Passed
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	✓