1 INTRODUCTION

In this coursework I have applied PITest[1] mutation testing tool on the provided *Apache Commons Collections* project source code. After running the PITest tool, at first, it mutates the original source code and creates a mutated version of the project. And then it runs the existing test suits on the mutated version to identify which mutants have been killed, survived and not covered by the original test suits. Finally, it generates a statistical report representing the mutation score by counting the number of killed, survived and not covered mutants. Utilizing this report, I explored which types of mutations have been killed, survived and not covered. Besides, I also identified which test cases were run during the mutation testing. Finally, I tried to extend the existing test suits to improve the coverage and mutation score. The rest of the report is organized as: Section 2 describes how I automated the build script to run the PITest tool. Section 3 provides the description of some killed mutants. Next, Section 4 explains how I improved the mutation score. At the end, Section 5 discuses how I increased the coverage and mutation score.

2 AUTOMATION

This section describes how I automated the build script to perform the mutation testing using the PITest tool. Depending on the build framework, PITest tool provides various options to utilize it. As the provided *Apache Commons Collections* source code is developed using the maven build framework, therefore I used the maven configuration options suggested by the official PITest website. To use this tool, I just added a plugin dependency inside the build section in the maven *pom.xml* configuration file. Listing 1 shows the plugin that I used. The guideline provided by the official PITest website is to use the LATEST version of this plugin. However, I used the version *1.4.11* instead of the LATEST as it is particularly recommend for the Apache Commons Collections project. Moreover, I also used Java 1.8 version to run my build script.

Listing 1: Added PITest Dependency To The build/plugins in the pom.xml

```
537:   <build>
538:       <defaultGoal>clean verify apache-rat:check clirr:check javadoc:javadoc</defaultGoal>
539:       <plugins>
538:           <plugin>
539:               <groupId>org.pitest</groupId>
540:               <artifactId>pitest-maven</artifactId>
541:               <version>1.4.11</version>
542:           </plugin>
....        </plugins>
....   </build>
```

By default PITest tool generates the output report in HTML format. However, if required this configuration can be changed. The official website suggests that to alter the report format an additional plugin need to be added under of report section in the *pom.xml* configuration file. I did not add this plugin in the report section of my build script as it is considered as optional and the default HTML report is very convenient to analyze. Then I run the PITest tool using a simple command *mvn org.pitest:pitest-maven:mutationCoverage*. It generates the report under the directory of *target/pit-reports*. Table 1 demonstrates the summary from the generated report. It shows the overall line coverage and mutation coverage performed by this tool.

Table 1: PITest Coverage Report for The Project Apache Commons Collections

Project Summary	Number of Classes	Line Coverage	Mutation Coverage
Apache Commons Collections	267	46% 6116/13211	42% (3532/8507)

[1]https://pitest.org/

3 KILLED MUTANTS

In this section, I described some mutants that are killed by the original test suits. After observing the report generated by the PITest tool, I have chosen a candidate code block from the source file *CompositeSet.java* that can be found under the package of *org.apache.commons.collections4.set*. Listing 2 represents the chosen code block. It shows a method namely *removeAll* that removes a collection of data from an instance of *CompositeSet*. Here, the number of mutants created for a particular line of code represented by a number besides the actual line number. And they are highlighted with green indicating that these mutants were successfully killed by the existing test suits. It is observed that 5 mutants were successfully killed in this single method.

Listing 2: Killed Mutants at *CompositeSet.java*

```
314      @Override
315      public boolean removeAll(final Collection<?> coll) {
316  1     if (CollectionUtils.isEmpty(coll)) {
317  1         return false;
318          }
319          boolean changed = false;
320          for (final Collection<E> item : all) {
321  1         changed |= item.removeAll(coll);
322          }
323  2       return changed;
324      }
```

Listing 3: Test Case for *removeAll* at *CompositeSetTest.java*

```
70:
71:
72:
73: public void testRemoveAll() {
74:     final CompositeSet<E> set = new CompositeSet<>
                        (new Set[]{ buildOne(), buildTwo() });
75:     assertFalse(set.removeAll(null));
76: }
77:
78:
79:
```

Table 2 shows the description of these killed mutants. It also provides the details explanation of how the mutated codes are created from the original source code. For example, at Line 316, the mutant *Negated Conditional* refers to changing the original condition by introducing a negation. Similarly, at Line 321 the original bitwise OR is replaced by a bitwise AND to create the mutant.

Many test methods are responsible to kill these mutants. Here I am going to describe one test method called *testRemoveAll* represented by Listing 3. I choose this test method because, it helps to kill some mutants in the *removeAll* method during the mutation testing. It is observed that at *Line:74* in Listing 3, the test method first creates a CompositeSet object and then it invokes the *removeAll* method by passing a argument value *null*. And then, it expects that the *removeAll* method should return false (Line:75). Now this test method easily killed the mutants such as *Replaced boolean return with true* at Line:317 and Line:323 in Listing 2. This is because, the test method always expects a false value (Line:75) in *assertFalse* statement. So when the mutants return true from Line:317 and Line:323 in Listing 2, the assertion fails in the test method and the test case cannot be passed. It infers that the test method successfully kills these mutants.

Table 2: Successfully Killed Mutants in *removeAll* at *CompositeSet.java*

Line	Mutants	Original Code	Mutated codes
316	Negated Conditional	`if(CollectionUtils.isEmpty(coll))`	`if(!CollectionUtils.isEmpty(coll))`
317	Replaced boolean return with true	`return false;`	`return true;`
321	Replaced bitwise OR with And	`changed |= item.removeAll(coll);`	`changed &= item.removeAll(coll);`
323	Replaced boolean return with false	`return changed;`	`return true;`
323	Replaced boolean return with true	`return changed;`	`return false;`

4 IMPROVING MUTATION SCORE

This section provides an exhaustive description of how I have improved the mutation score by updating an existing test case in test suits. Firstly, after analyzing the PITest report, I have identified a method block where a few mutants have not been killed but mutated lines have been covered by the original test suits. Listing 4 represents that selected method block named *removeComposited* which is located in the source file *CompositeMap.java* under the package of *org.apache.commons.collections4.map*. This method can be invoked from an instance of a *CompositeMap* where it takes a new map and remove the values of that map from its own instance. The corresponding test case for testing this method is illustrated in Listing 5.

In the Listing 4, the number of mutants are represent by a number beside the actual line number. The highlighted green lines represents the killed mutants and the highlighted red lines indicates the survived mutants. It is found that in total 11 mutants are introduced in this method. All the mutants are covered where only 7 mutants have been killed by the existing test case and the rest of the 4 are survived mutants. It means the existing test suit cannot killed these 4 mutants although these mutants are covered. Table 3 provides these 11 mutants with the description, how these are obtained and their status.

Listing 4: Survived Mutants in *removeComposited*

```
160:
161:  public synchronized Map<K,V> removeComposited(final Map<K,V> map){
162:    final int size = this.composite.length;
163: 3    for (int i = 0; i < size; ++i) {
164: 1      if (this.composite[i].equals(map)) {
165: 1        final Map<K, V>[] temp = new Map[size - 1];
166: 1        System.arraycopy(this.composite, 0, temp, 0, i);
167: 4        System.arraycopy(this.composite,i + 1,temp,i,size-i-1);
168:         this.composite = temp;
169: 1        return map;
170:       }
171:     }
172:      return null;
173:   }
```

Listing 5: Original JUnit Test Case for *removeComposited*

```
90: public void testRemoveComposited() {
91:    final CompositeMap<K,V> map =new CompositeMap<>
                                    (buildOne(),buildTwo());
92:    final HashMap<K, V> three = new HashMap<>();
93:    three.put((K) "5", (V) "five");
94:    map.addComposited(null);
95:    map.addComposited(three);
96:    assertTrue(map.containsKey("5"));
97:    map.removeComposited(three);
98:    assertFalse(map.containsKey("5"));
99:    map.removeComposited(buildOne());
100:   assertFalse(map.containsKey("2"));
101:
102: }
```

To validate the capability of the *removeComposited* method, the existing test case only adds a new map to the composite map and checks whether the composite map contains these new map values (Listing 5; Line:91, 92 and 93). After that, it removes this map and checks again the values are removed or not (Listing 5; Line:96, 98 and 100). There are many reasons why the existing test case cannot kill those mutants. For example, it did not check the boundary condition when we want to remove a map that's values did not exist inside the instance of *CompositeMap*. Then, in the test case, it is not asserted that if a map is removed from the top, middle or from the end of the instance of *CompositeMap* how the instance organizes itself to represent the values.

Table 3: Survived and Killed Mutants in `removeComposited`

Line	Mutatns	Original Code	Mutated Code	Status
163	Changed conditional boundary	`i <size`	`i <= size`	Survived
163	Change increment from 1 to -1	`++i`	`--i`	Killed
163	Negated conditional	`i <size`	`i >= size`	Killed
164	Negated conditional	`if (this.composite..)`	`if (!this.composite..)`	Killed
165	Replaced integer subtraction with addition	`....= new Map[size - 1];`	`.....= new Map[size + 1];`	Killed
166	Remove call System.arraycopy()	`System.arraycopy(...);`	`//System.arraycopy(...);`	Killed
167	Replaced integer addition with subtraction	`(this.composite, i + 1,...)`	`(this.composite, i - 1, ...)`	Survived
167	Remove call System.arraycopy()	`System.arraycopy(...)`	`//System.arraycopy(...)`	Survived
167	Replaced integer subtraction with addition	`(..., temp, i, size - i - 1)`	`(...size + i - 1)`	Killed
167	Replaced integer subtraction with addition	`(..., temp, i, size - i - 1)`	`(...size - i + 1)`	Killed
169	Replace return value with null	`return map;`	`return null;`	Survived

The coursework requirement is to kill one survived mutant, but to improve the mutation score, killing one mutants would not help. Therefore, I decided to kill all 4 survived mutants by modifying the existing test case. Listing 6 represents the modified version of the existing test case *testRemoveComposited*. In the test case, first an object of *CompositeMap* has been instantiated using two new maps. Then, I added some new lines of code in this test case by creating an new map *"dummy"*(Listing 6;Line:96-98) and tried to remove the values of this map from the composite map although the composite map did not contain any values from the map *"dummy"*. Next, I added *dummy* into the *CompositeMap* and removed those two maps that were previously added while initializing the *CompositeMap* (Listing 6;Line:100-102). After that, I removed a map from the top and check whether any value presents in the map after removing. And then I also removed another map at the end and do the same checking. This imposed the instance of the *CompositeMap* reorganizes itself to represent the values. After, implementing the test case in this way, it helped to killed all the surviving mutants. The yellow highlighted

lines in the Listing 6 demonstrate the newly added lines that improves the test case. Then I run the PITest tool again to verify weather the improved test case can kill these mutants. Listing 7 shows the status of mutation test report for the method *removeComposited*. It is observed that all the lines are highlighted with green that indicates all these mutants are killed because of the improved test case.

Listing 6: Improved Unit Test for *removeComposited*

```
90: public void testRemoveComposited() {
91:     final CompositeMap<K,V> map = new CompositeMap<>
                                (buildOne(),buildTwo());
92:     final HashMap<K, V> three = new HashMap<>();
93:     three.put((K) "5", (V) "five");
94:     map.addComposited(null);
95:     map.addComposited(three);
96:     final HashMap<K, V> dummy = new HashMap<>();
97:     dummy.put((K) "d", (V) "dummy");
98:     assertEquals(null, map.removeComposited(dummy));
99:     map.addComposited(dummy);
100:    map.removeComposited(buildOne());
101:    assertFalse(map.containsKey("2"));
102:    assertEquals(three, map.removeComposited(three));
103:    assertFalse(map.containsKey("5"));
104: }
```

Listing 7: Killed Mutants in *removeComposited*

```
160:
161:  public synchronized Map<K,V> removeComposited(final Map<K,V> map){
162:    final int size = this.composite.length;
163: 3   for (int i = 0; i < size; ++i) {
164: 1      if (this.composite[i].equals(map)) {
165: 1         final Map<K, V>[] temp = new Map[size - 1];
166: 1         System.arraycopy(this.composite, 0, temp, 0, i);
167: 4         System.arraycopy(this.composite,i + 1,temp,i,size-i-1);
168:          this.composite = temp;
169: 1         return map;
170:       }
171:     }
172:      return null;
173:   }
174:
175:
```

This improved test case just marginally increases the mutation score. For example, Table 4 illustrates how the mutation score increases after the improvement. It compares the score before and after improvement of the test case. The overall score is represented in form of percentage (e.g; the number of killed mutants/total number of mutants). Since the improved test case can kill the 4 survived mutants, therefore, the number of killed mutants is increased by 4. The mutation killing first occurs at class level and increases the score. And then the score also increases the score of its the upper level such as package level and project level. Before the improvement of the test case the number of killed mutants are 94 in the class *CompositeMap.java*. The improved test case killed the 4 survived mutants and the number of killed mutants becomes 98. In term of percentage, this increment would not increase the overall score of the whole projects, however, it increases the total number of killed mutants. As we see before improving the test case the total number of killed mutants is 3232 and after modifying the test case now it becomes 3536 although the overall mutation coverage score remain 42%.

Table 4: Mutation Score Improvement in *CompositeMap.java*

No	Summary	Name	Line Coverage	Mutation Coverage
1	Class Summary	CompositeMap.java (Before)	95% (87/92)	95% (94/99)
		CompositeMap.java (After Improvement)	96% (88/92)	99% (98/99)
2	Package Summary	org.apache.commons.collections4.map (Before)	30% (972/3197)	24% (480/1990)
		org.apache.commons.collections4.map (After Improvement)	30% (973/3197)	24% (484/1990)
3	Project Summary	Apache Common Collestions (Before)	46% (6116/13211)	42% (3532/8507)
		Apache Common Collections (After Improvement)	46% (6117/13211)	42% (3536/8507)

5 IMPROVING COVERAGE AND MUTATION SCORE

In this section, I discuss how I have improved the mutation coverage score including the mutation score. After analyzing the PITest report, I identified a code block that contains mutants which are not covered by the existing test suits. Hence, these mutants are not killed by the original test cases. Listing 8 represents that selected code block. It demonstrates a method named *putAll* in the source file *ListOrderedMap.java* which can be found in the package *org.apache.commons.collections4.map*. The method *putAll* can be invoked from an instance of *ListOrderedMap*. The instance can takes an index and a new map as the arguments of *putAll*. And then, the instance adds all the elements of the new map into its own map.

The number of mutants in this method are shown by a number besides the actual line number and the highlighted red lines represents that these lines are not covered by the test suits. Table 5 describes these mutants, how they are created from the original code and their status. In total 7 mutants are created in this method during the mutation test where none of these mutants were covered.

Listing 8: Not Covered Mutants in *putAll* at *ListOrderedMap.java*

```
254     public void putAll(int index, final Map<? extends K, ? extends V> map) {
255  4      if (index < 0 || index > insertOrder.size()) {
256             throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + insertOrder.size());
257         }
258         for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {
259             final K key = entry.getKey();
260             final boolean contains = containsKey(key);
261             // The return value of put is null if the key did not exist OR the value was null
262             // so it cannot be used to determine whether the key was added
263             put(index, entry.getKey(), entry.getValue());
264  1          if (!contains) {
265                 // if no key was replaced, increment the index
266  1              index++;
267             } else {
268                     // otherwise put the next item after the currently inserted key
269  1                  index = indexOf(entry.getKey()) + 1;
270             }
271         }
272     }
```

Now, in order to cover and kill these mutants, I need to update the test cases that may responsible for testing this method. I searched for finding the test cases that were developed to test this *putAll* method for the instance of *ListOrderedMap*. However, I could not find any test case that invoked this method from the instance of *ListOrderedMap* for testing purpose. Therefore, I have written a new test case to validate this method capability. And I also placed the new test case in the test class *MapUtilTest.java*. The reason for choosing this source file for writing the new test is that *MapUtilTest.java* test class contains numerous test cases to validate various types of *Map*. And for testing Map related methods, the PITest tools run all these test cases of this test class during the mutation testing.

Table 5: Not Covered Mutants in `putALL`

Line	Mutatns	Original Code	Mutated Code	Status
255	Changed conditional boundary	`if(index<0||...)`	`if(index <=0 ||...)`	No Coverage
255	Changed conditional boundary	`if(...||index>insertOrder.size())`	`if(...||index >=insertOrder.size())`	No Coverage
255	Negated conditional	`if(index<0||...)`	`if(index>= 0 || ...`	No Coverage
255	Negated conditional	`if(...||index >insertOrder.size())`	`if(...|| index<=insertOrder.size())`	No Coverage
264	Negated conditional	`if(!contains)`	`if(contains)`	No Coverage
266	Change increment from 1 to -1	`index++;`	`index-;`	No Coverage
269	Replaced integer addition with subtraction	`index=indexOf(entry.getKey())+1`	`index = indexOf(entry.getKey())-1`	No Coverage

Listing 9 shows the new JUnit test case that I have written for covering and killing these mutants. From Table 5, it is examined that, among the 7 mutants 5 mutants are conditional boundary and condition negation related. It implies that to cover and kill these mutants, the boundary values should be validated by the new test case. Therefore I started witting the test case by creating a map named *mapOne* with 3 values (Listing 9; Line:1158-1160). Then I created an instance of *ListOrderMap* called *orderMapOne* that takes the *mapOne* in it constructor (Listing 9; Line:1161-1162). As expected, the size of the *orderMapOne* is now 3. So the boundary values for this *orderMapOne* is 0 and 3. After that, I created two new maps called *mapTwo* and *mapThree*. Here *mapTwo* contains 2 elements that are already presented in the *orderMapOne*. And *mapThree* also contains 2 elements which are not presented in the *orderMapOne*. Now I put all elements of the *mapTwo* at the index of 0 in the *orderMapOne* and put all the elements of *mapThree* at the index of 3 by invoking its *putAll* method (Listing 9; Line:1163-1166). Then I asserted to confirm that the values are placed in the right order of this *orderMapOne*. Now, the size of the *orderMapOne* becomes 5.

Listing 9: New Unit Test Case for *putAll* in *MapUtilTest.java*

```
1156 @Test
1157       public void testListOrderedMap() {
1158       Map<String, String> mapOne = new HashMap<String, String>() {{ put("k1", "value1");put("k2", "value2");
1159           put("k3", "value3");
1160       }};
1161       ListOrderedMap<String, String> orderMapOne = ListOrderedMap.listOrderedMap(mapOne);
1162       assertTrue("returned object should be a OrderedMap", orderMapOne instanceof ListOrderedMap);
1163       Map<String, String> mapTwo = new HashMap<String, String>() {{ put("k1", "v1");put("k2", "v2"); }};
1164       orderMapOne.putAll(0, mapTwo);
1165       Map<String, String> mapThree = new HashMap<String, String>() {{ put("key1", "V1");put("key2", "V2"); }};
1166       orderMapOne.putAll(3, mapThree);
1167       assertEquals("k3", orderMapOne.get(2));
1168       assertEquals("key1", orderMapOne.get(3));
1169       Map<String, String> mapFour = new HashMap<String, String>() {{ put("kv1", "KV1");put("key1", "V1"); }};
1170       orderMapOne.putAll(5, mapFour);
1171       assertEquals(6, orderMapOne.size());
1172   }
```

At last, I created a new map called *mapFour* that contains two values where one value already presented in *orderMap* and another element in completely new. Again I added the *mapFour* at boundary index 5. Thus, it only put the single new element in to the *orderMapOne* and the size of the *orderMapOne* is now 6 (Listing 9; Line:1169-1171). By putting new maps into the boundary position of the *ListOrderMap*, helps to cover and kill all the conditional boundary related mutants. And putting various combination of maps covered and killed the other mutants, such as *Negated Conditional*, *Changed increment from 1 to -1* and *Replaced addition with subtraction.*

Listing 10: Covered and Killed Mutants in the method *putAll* in *ListOrderedMap.java*

```
254     public void putAll(int index, final Map<? extends K, ? extends V> map) {
255  4      if (index < 0 || index > insertOrder.size()) {
256             throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + insertOrder.size());
257         }
258         for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {
259             final K key = entry.getKey();
260             final boolean contains = containsKey(key);
261             // The return value of put is null if the key did not exist OR the value was null
262             // so it cannot be used to determine whether the key was added
263             put(index, entry.getKey(), entry.getValue());
264  1          if (!contains) {
265                 // if no key was replaced, increment the index
266  1              index++;
267             } else {
268                    // otherwise put the next item after the currently inserted key
269  1                 index = indexOf(entry.getKey()) + 1;
270             }
271         }
272     }
```

Now I run the PITest on the project in order to analyze how this newly added test case change the coverage and mutation score. Listing 10 shows the method block of *putAll* in the PITest newly generated report. It is observed that the lines are highlighted with green indicating that these mutants are covered and killed. Table 6 shows the summary of result derived from the newly generated PITest report. It is observed that, in the class level the line coverage score increases from 56% to 69% and the mutation coverage score rises to 65% from 45%. Consequently, it results in 1% increase in the score of package level although the overall project level score remain same. The overall score in term of percentage would not change, however, the number of line coverage and the number of killed mutants are increased.

Table 6: Improved Coverage and Mutation Score for `putALL`

No	Summary	Name	Line Coverage	Mutation Coverage
1	Class Summary	ListOrderedMap.java (Before)	56% (111/200)	45% (54/121)
		ListOrderedMap.java (After Improvement)	69% (137/200)	60% (72/121)
2	Package Summary	org.apache.commons.collections4.map (Before)	30% (973/3197)	24% (484/1990)
		org.apache.commons.collections4.map (After Improvement)	31% (999/3197)	25% 502/1990
3	Project Summary	Apache Common Collections (Before)	46% (6117/13211)	42% (3536/8507)
		Apache Common Collections (After Improvement)	46% (6143/13211)	42% (3554/8507)