Toronto Metropolitan University

Department of Electrical & Computer Engineering
Faculty of Engineering & Architectural Science

Course Title:	Software Testing and QA
Course Number:	COE891
Semester/Year	W2025

Instructor	Dr. Reza Samavi

Assignment/Lab Title:	Final Project

Submission Date:	Apr 10, 2025

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Asif	Eeman	501065412	03	E.A.
Bhuiyan	Sabia	500833949	03	S.B.
Labeeba	Maria	500960386	03	M.L
Maudiwala	Husain Yunus	501010811	03	H.M.
Mirza	Ariba	501029045	03	A.M.

*By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: http://www.ryerson.ca/senate/current/pol60.pdf

Table of Contents

Introduction

This project focused on the practical application of unit testing techniques taught throughout the course, with the goal of strengthening our understanding of software testing principles and their real-world implementation. To achieve this, the open-source Java application **SmartTube**, developed by GitHub user *yuliskov*, was selected as the system under test. SmartTube provided a complex, real-world codebase that allowed for meaningful testing challenges and deeper insight into code quality analysis.

Ten methods were selected from various classes within the application, representing a diverse range of functionalities. These methods were rigorously tested using four core testing abstractions covered in class: **Input Space Partitioning (ISP)** to identify representative input classes, **graph-based testing** through the construction of **Control Flow Graphs (CFG)** and **Data Flow Graphs (DFG)** to evaluate logical paths and data usage, **logic-based testing** to assess condition coverage and decision-making structures, and **mutation testing** to measure the effectiveness of the test cases by introducing small code changes (mutants).

This report presents the tools and technologies utilized throughout the project, such as Eclipse IDE, JUnit, and PIT for mutation testing. It also highlights the challenges encountered, key findings, and overall learning outcomes gained from working with a production-level Android Java application. The project ultimately provided valuable hands-on experience in designing test cases and understanding the importance of thorough testing in building reliable, maintainable software.

Testing Requirements

SmartTube GitHub repository: https://github.com/yuliskov/SmartTube

The testing was carried out in the Eclipse Java IDE, using the JUnit testing framework along with the PitClipse plugin for mutation testing. The SharedModules repository contains functionalities that are utilized within SmartTube, as a result, it was automatically initialized when the SmartTube repository was cloned.

Methods Under Test

Table 1. Methods and Classes that were Tested in the SmartTube System

Class	Methods	Path
CopyOnWriteHashList.java	1. public boolean add(T item) 2. public void add(int index, T item)	common/src/main/java/com/liskovsoft/smartyoutubetv2/common/utils/**CopyOnWriteHashList.java** **https://github.com/yuliskov/SmartTube/blob/master/common/src/main/java/com/liskovsoft/smart**

		youtubetv2/common/utils/CopyOnWriteHashList.java
PasswdURI.java	3. public String getUsername() 4. public String getPassword()	common/src/main/java/com/liskovsoft/smartyoutubetv2/common/proxy/**PasswdURI.java** https://github.com/yuliskov/SmartTube/blob/master/common/src/main/java/com/liskovsoft/smartyoutubetv2/common/proxy/PasswdURI.java
DateFormatter.java	5. public static String format(Date date, String format) 6. public static boolean isSameDay(Date date1, Date date2)	chatkit/src/main/java/com/stfalcon/chatkit/utils/**DateFormatter.java** https://github.com/yuliskov/SmartTube/blob/master/chatkit/src/main/java/com/stfalcon/chatkit/utils/DateFormatter.java
Proxy.java	7. public Proxy(Type type, SocketAddress sa) 8. public final boolean equals(Object obj)	common/src/main/java/com/liskovsoft/smartyoutubetv2/common/proxy/**Proxy.java** https://github.com/yuliskov/SmartTube/blob/master/common/src/main/java/com/liskovsoft/smartyoutubetv2/common/proxy/Proxy.java
ActionHelpers.java	9. public static int getIconHighlightColor(Context context) 10. public static int getIconGrayedOutColor(Context context)	smarttubetv/src/main/java/com/liskovsoft/smartyoutubetv2/tv/ui/playback/actions/**ActionHelpers.java** https://github.com/yuliskov/SmartTube/blob/master/smarttubetv/src/main/java/com/liskovsoft/smartyoutubetv2/tv/ui/playback/actions/ActionHelpers.java

The table above has the following classes and their relative path in the GitHub repository were used along with methods.

Testing

The following examples contain some of the methods tested on using the four testing approaches.

Test Approach #1 - Input Space Partitioning:

Input Space Partitioning is a mathematical approach to determine what tests should be written and when to stop writing tests. Classes are modelled as functions where potential inputs are tested, and this range of possible inputs (input space) can be divided into several subsets, known as partitions. Here, each partition represents a similar input behaviour, for example, all positive integers, meaning one value from each set will suffice. This way, all within a partition can be tested using one input allowing for more efficient testing while still reaching maximum coverage. The steps for ISP include identifying the inputs, partitioning the input space, selecting test cases to represent each partition and writing and executing the tests.

Example #1:

The *CopyOnWriteHashList* class extends *CopyOnWriteArrayList* to enforce uniqueness while retaining thread-safety and performance benefits suited for read-heavy scenarios. It overrides the *add(T item)* and *add(int index, T item)* methods to prevent duplicate entries by removing existing items before adding new ones. The *add(T item)* method returns false if the item is null or already at the last index, otherwise replacing duplicates before adding. The *add(int index, T item)* method either inserts the item at the specified index or appends it if the index is invalid, while also removing duplicates before insertion. Unlike *CopyOnWriteArrayList*, this class ensures items are not duplicated within the list, offering a thread-safe, order-preserving, and duplicate-free collection.

```
public class CopyOnWriteHashList<T> extends CopyOnWriteArrayList<T> {
    @Override
    public boolean add(T item) {
        int index = size() - 1;
        if (item == null || (index >= 0 && indexOf(item) == index)) {
            return false;
        } else if (contains(item)) {
            remove(item);
        }

        return super.add(item);
    }
```

Figure 1 : Method add(T item)

```java
    @Override
    public void add(int index, T item) {
        if (item == null || (index >= 0 && indexOf(item) == index)) {
            return;
        } else if (contains(item)) {
            remove(item);
        }

        if (index >= 0 && index < size()) {
            super.add(index, item);
        } else {
            super.add(item);
        }
    }
}
```

Figure 2 : Method add(int index, T item)

The test cases for the classes described above were determined by checking the input parameters, creating the partitions, and writing the corresponding tests. Tables 2 and 3 show the input space for each method, which are then used to determine the partitions.

Table 2. Input space for add(T item) method

Input Parameter	Valid Input Values	Invalid Input Values
item	Non-null, unique item	Null item
	Non-null, duplicate item	Duplicate item at last index
	Null item	-

Partitions for add(T item) Method

Valid Partitions

1. **Input:** item = Non-null, unique item
   Expected Output: Successful addition, returns true.
2. **Input:** item = Non-null, duplicate item (exists elsewhere in the list)
   Expected Output: Successful replacement, returns true (removes existing instance and adds the new one).

Invalid Partitions

1. **Input:** item = null
   Expected Output: Item is not added, returns false.
2. **Input:** item = Non-null, duplicate item located at the last index
   Expected Output: Item is not added, returns false.

Table 3. Input space for add(int index, T item) method

Input Parameter	Valid Input Values	Invalid Input Values
Index, item	Valid index, unique item	Null item
	Valid index, duplicate item	Negative index
	Invalid index, unique item	Index out of bounds
	Invalid index, duplicate item	-

Partitions for add(int index, T item) Method

Valid Partitions

1. **Input:** index = Valid index within bounds, item = Non-null, unique item
   Expected Output: Successful addition at specified index, returns true.
2. **Input:** index = Valid index within bounds, item = Non-null, duplicate item (exists elsewhere in the list)
   Expected Output: Successful replacement, returns true (removes existing instance and inserts the new one at the specified index).
3. **Input:** index = Invalid index (out of bounds, greater than size), item = Non-null, unique item
   Expected Output: Successful addition at the end of the list, returns true.
4. **Input:** index = Invalid index (out of bounds, greater than size), item = Non-null, duplicate item (exists elsewhere in the list)
   Expected Output: Successful replacement, returns true (removes existing instance and adds the new one to the end of the list).

Invalid Partitions

1. **Input:** item = null, index = Any value (valid or invalid)
   Expected Output: Item is not added, returns immediately, no change to the list.
2. **Input:** index = Negative index, item = Non-null
   Expected Output: Item is not added, returns immediately, no change to the list.

The code along with its successful test run can be seen below:

```java
package ISP1;


import org.junit.Before;

public class CopyOnWriteHashListTest {
    private CopyOnWriteHashList<String> list;

    @Before
    public void setUp() {
        list = new CopyOnWriteHashList<>();
    }

    @Test
    public void testAdd_nullItem_shouldNotBeAdded() {
        boolean result = list.add(null);
        assertFalse(result);
        assertTrue(list.isEmpty());
    }

    @Test
    public void testAdd_uniqueItem_shouldBeAdded() {
        boolean result = list.add("Apple");
        assertTrue(result);
        assertEquals(1, list.size());
        assertEquals("Apple", list.get(0));
    }

    @Test
    public void testAdd_duplicateAtEnd_shouldNotAddAgain() {
        list.add("Banana");
        boolean result = list.add("Banana"); // should be ignored because it's at end
        assertFalse(result);
        assertEquals(1, list.size());
    }

    @Test
    public void testAdd_duplicateNotAtEnd_shouldBeMovedToEnd() {
        list.add("A");
        list.add("B");
        list.add("C");

        boolean result = list.add("B"); // should move B to end
        assertTrue(result);
        assertEquals(3, list.size());
        assertEquals("C", list.get(1));
        assertEquals("B", list.get(2)); // B now at end
```

```java
    @Test
    public void testAddIndex_validIndex_uniqueItem_shouldAddAtIndex() {
        list.add("X");
        list.add("Y");

        list.add(1, "Z");

        assertEquals(3, list.size());
        assertEquals("X", list.get(0));
        assertEquals("Z", list.get(1));
        assertEquals("Y", list.get(2));
    }

    @Test
    public void testAddIndex_invalidIndex_shouldAppend() {
        list.add("One");
        list.add(100, "Two"); // out of bounds, so should append

        assertEquals(2, list.size());
        assertEquals("Two", list.get(1));
    }

    @Test
    public void testAddIndex_duplicateAtSameIndex_shouldNotAddAgain() {
        list.add("First");
        list.add(0, "First"); // same index

        assertEquals(1, list.size());
        assertEquals("First", list.get(0));
    }

    @Test
    public void testAddIndex_duplicateNotAtIndex_shouldMoveToIndex() {
        list.add("One");
        list.add("Two");
        list.add(0, "Two"); // remove and insert at 0

        assertEquals(2, list.size());
        assertEquals("Two", list.get(0));
        assertEquals("One", list.get(1));
    }
}

```

Figure 3 : Code for CopyOnWriteHashListTest.java

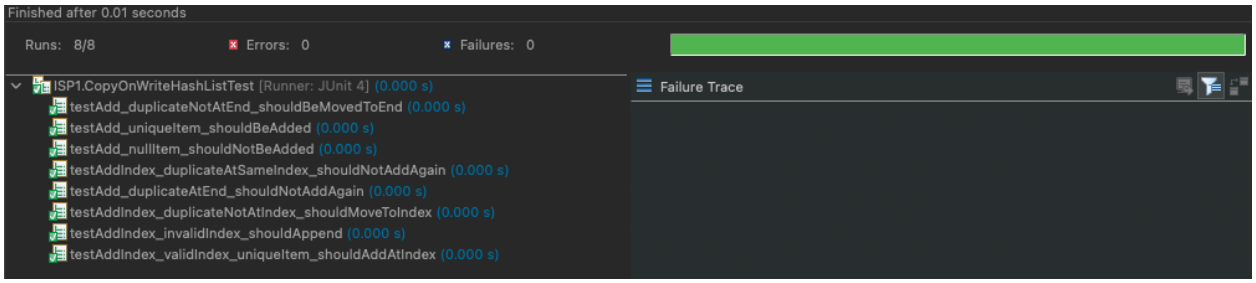

Figure 4: Successful test run

Example #2:

The second example focused on the constructor method *public Proxy(Type type, SocketAddress sa)* and the *public final boolean equals(Object obj)* method of the Proxy class. The purpose of this class was to handle the network proxy settings in a custom manner, by allowing the application to define the proxy types, like HTTP/SOCK/DIRECT, and associate them with a specific network address. The constructor creates a new instance of a Proxy object to define how network connections can be made. The equals method overrides the built-in *Object.equals()* method to enable comparison of two Proxy objects based on their contents as opposed to their memory address. The tested methods and their explanations are detailed in Figures 5 and 6 below.

```
/**
 * Creates an entry representing a PROXY connection.
 * Certain combinations are illegal. For instance, for types Http, and
 * Socks, a SocketAddress <b>must</b> be provided.
 * <P>
 * Use the {@code Proxy.NO_PROXY} constant
 * for representing a direct connection.
 *
 * @param type the {@code Type} of the proxy
 * @param sa the {@code SocketAddress} for that proxy
 * @throws IllegalArgumentException when the type and the address are
 * incompatible
 */
public Proxy(Type type, SocketAddress sa) {
    if ((type == Type.DIRECT) || !(sa instanceof PasswdInetSocketAddress))
        throw new IllegalArgumentException("type " + type + " is not compatible with address " + sa);
    this.type = type;
    this.sa = sa;
}
```

Figure 5. Constructor method Proxy and its description.

```
    /**
     * Compares this object against the specified object.
     * The result is {@code true} if and only if the argument is
     * not {@code null} and it represents the same proxy as
     * this object.
     * <p>
     * Two instances of {@code Proxy} represent the same
     * address if both the SocketAddresses and type are equal.
     *
     * @param   obj   the object to compare against.
     * @return  {@code true} if the objects are the same;
     *          {@code false} otherwise.
     * @see InetSocketAddress#equals(Object)
     */
    public final boolean equals(Object obj) {
        if (obj == null || !(obj instanceof Proxy))
            return false;
        Proxy p = (Proxy) obj;
        if (p.type() == type()) {
            if (address() == null) {
                return (p.address() == null);
            } else
                return address().equals(p.address());
        }
        return false;
    }
```

Figure 6. Comparison method equals and its description.

The test cases for the classes described above were determined by checking the input parameters, creating the partitions, and writing the corresponding tests. Tables 4 and 5 show the input space for each method, which are then used to determine the partitions.

Table 4. Input space for Proxy method

Input Parameter	Valid Input Values	Invalid Input Values
Type type	DIRECT	
	HTTP	
	SOCKS	
SocketAddress sa	Instance of PasswdInetSocketAddress	Cannot be null if type is HTTP or SOCK

From this table, the following partitions can be derived:
Valid:

- Input: type = HTTP, sa = valid PasswdInetSocketAddress
  - Expected Output: Successful creation
- Input: type = SOCKS, sa = valid PasswdInetSocketAddress
  - Expected Output: Successful creation

Invalid:

- Input: type = DIRECT, sa = valid PasswdInetSocketAddress
  - Expected Output: IllegalArgumentException
- Input: type = null, sa = valid PasswdInetSocketAddress
  - Expected Output: Successful creation (constructor does not check for null)

The code for these tests is provided in Figure 7 in the *testProxyConstructor_validHTTPProxy*, *testProxyConstructor_validSOCKSProxy*, *testProxyConstructor_invalidDIRECTProxy*, and the *testProxyConstructor_nullTypeAllowed* methods. The successful run of these tests can be seen in Figure 8 below.

Similarly, the tests for the equals method were determined as follows.

Table 5. Input space for equals method

Input Parameter	Valid Input Values	Invalid Input Values
Object obj (compared against)	Any object (including null)	

From this table, the following partitions can be derived:
Valid:

- Input: obj with the same type and same sa (socket address)

    - Expected Output: true

Invalid:

- Input: obj with a different type and the same sa
    - Expected Output: false
- Input: obj with the same type and a different sa
    - Expected Output: false
- Input: obj is null
    - Expected Output: false
- Input: obj is not a Proxy instance (can't use it to compare)
    - Expected Output: false

The code for these tests is provided in Figure 7 in the *testEquals_sameProxyObjects*, *testEquals_differentType*, *testEquals_differentSocketAddress*, *testEquals_withNullObject*, and the *testEquals_withNonProxyObject* methods. The successful run of these tests can be seen in Figure 8 below.

```
1  package ISP3;
2
3  import org.junit.Test;
7
8  /**
9   * Unit tests for the Proxy class to verify constructor and equals() method behavior
10  * based on Input Space Partitioning (ISP) testing.
11  */
12 public class ProxyTest{
13
14     /**
15      * Test case: Valid HTTP Proxy with a valid PasswdInetSocketAddress
16      * Expected result: Successful creation of Proxy instance
17      */
18     @Test
19     public void testProxyConstructor_validHTTPProxy() {
20         SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
21         Proxy proxy = new Proxy(Proxy.Type.HTTP, sa);
22         assertEquals(Proxy.Type.HTTP, proxy.type());
23         assertEquals(sa, proxy.address());
24     }
25
26     /**
27      * Test case: Valid SOCKS Proxy with a valid PasswdInetSocketAddress
28      * Expected result: Successful creation of Proxy instance
29      */
30     @Test
31     public void testProxyConstructor_validSOCKSProxy() {
32         SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 1080, "user", "pass");
33         Proxy proxy = new Proxy(Proxy.Type.SOCKS, sa);
34         assertEquals(Proxy.Type.SOCKS, proxy.type());
35         assertEquals(sa, proxy.address());
36     }
37
38     /**
39      * Test case: Invalid Direct Proxy with a non-null SocketAddress
40      * Expected result: IllegalArgumentException
41      */
42     @Test
43     public void testProxyConstructor_invalidDirectProxy() {
44         SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
45         assertThrows(IllegalArgumentException.class, () -> new Proxy(Proxy.Type.DIRECT, sa));
46     }
47
48     /**
49      * Test case: Comparing two Proxy objects with same type and same address
50      * Expected result: true
51      */
```

```
52      @Test
53      public void testEquals_sameProxyObjects() {
54          SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
55          Proxy p1 = new Proxy(Proxy.Type.HTTP, sa);
56          Proxy p2 = new Proxy(Proxy.Type.HTTP, sa);
57          assertTrue(p1.equals(p2));
58      }
59
60      /**
61       * Test case: Passing null as Proxy.Type should still construct Proxy
62       * since there is no null check and condition evaluates safely.
63       */
64      @Test
65      public void testProxyConstructor_nullTypeAllowed() {
66          SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
67          Proxy proxy = new Proxy(null, sa);
68          assertNull(proxy.type());
69          assertEquals(sa, proxy.address());
70      }
71
72      /**
73       * Test case: Comparing two Proxy objects with different types
74       * Expected result: false
75       */
76      @Test
77      public void testEquals_differentType() {
78          SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
79          Proxy p1 = new Proxy(Proxy.Type.HTTP, sa);
80          Proxy p2 = new Proxy(Proxy.Type.SOCKS, sa);
81          assertFalse(p1.equals(p2));
82      }
83
84      /**
85       * Test case: Comparing two Proxy objects with same type but different addresses
86       * Expected result: false
87       */
88      @Test
89      public void testEquals_differentSocketAddress() {
90          SocketAddress sa1 = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
91          SocketAddress sa2 = PasswdInetSocketAddress.createUnresolved("localhost", 9090, "user", "pass");
92          Proxy p1 = new Proxy(Proxy.Type.HTTP, sa1);
93          Proxy p2 = new Proxy(Proxy.Type.HTTP, sa2);
94          assertFalse(p1.equals(p2));
95      }
96
97      /**
98       * Test case: Comparing a Proxy object to null
99       * Expected result: false
100      */
101     @Test
102     public void testEquals_withNullObject() {
103         SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
104         Proxy p1 = new Proxy(Proxy.Type.HTTP, sa);
105         assertFalse(p1.equals(null));
106     }
107
108     /**
109      * Test case: Comparing a Proxy object to an unrelated object type
110      * Expected result: false
111      */
112     @Test
113     public void testEquals_withNonProxyObject() {
114         SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
115         Proxy p1 = new Proxy(Proxy.Type.HTTP, sa);
116         Object obj = new Object();
117         assertFalse(p1.equals(obj));
118     }
119 }
120
```

Figure 7. Code for ProxyTest.java

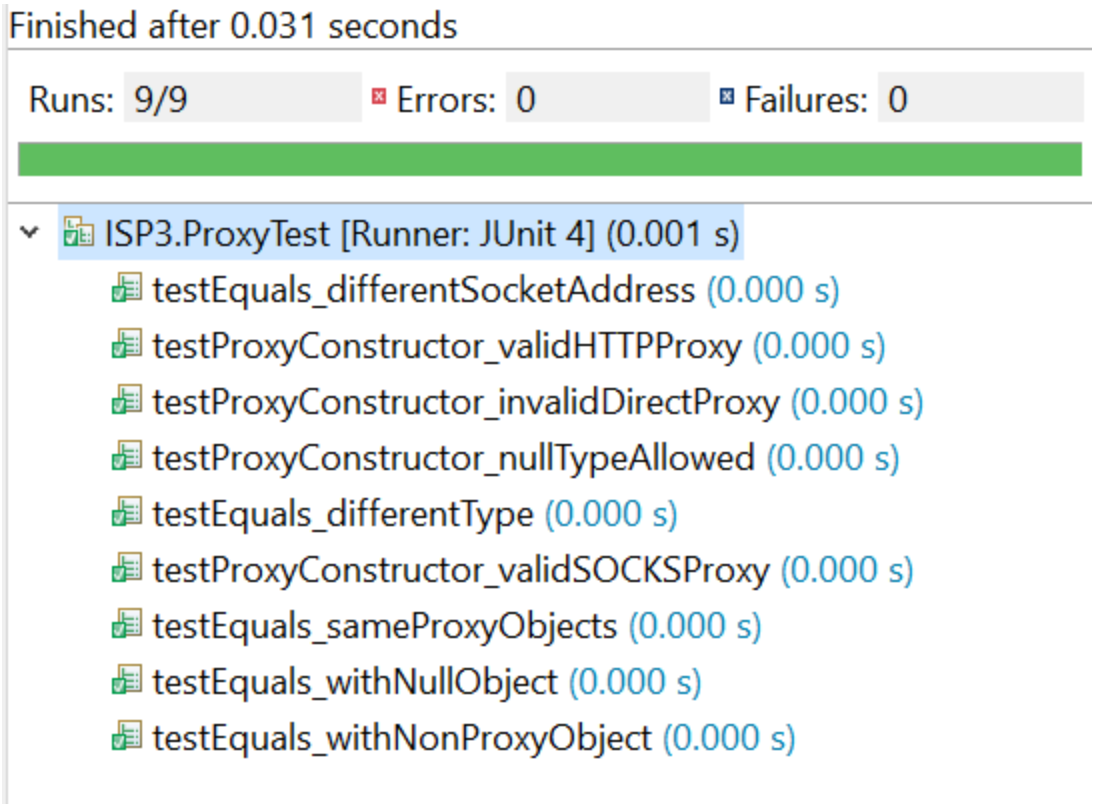

Figure 8. Successful test execution for ProxyTests.java

Test Approach #2 - Graph-based testing (CFG and DFG):

This approach required the analysis of the control and data flow for the following examples by creating Control Flow Graphs (CFG's) and Data Flow Graphs (DFG's). This was followed by implementing JUnit test cases to test node coverage for the CFG and use coverage for DFG. Node coverage ensures that every node/statement in the CFG is executed at least once during testing. This is important as it confirms that all parts of the method are reachable. Use coverage focuses on the data flow of the method, by ensuring that every variable use is applied during testing. This ensures that variables are correctly used and consistent throughout the code and can help find uninitialized variables. Overall, this form of testing allows a more visual approach to analysing different executing paths and variable dependencies for each method.

Example #1:

The method *public boolean add(T item)* belonging to the java class *CopyOnWriteHashList.java* java was tested as seen in Figure 9.

```java
public boolean add(T item) {
    int index = size() - 1;
    if (item == null || (index >= 0 && indexOf(item) == index)) {
        return false;
    } else if (contains(item)) {
        remove(item);
    }

    return super.add(item);
}
```

Figure 9. Method under test for graph based testing

For this method, the following CFG seen in Figure 10 was designed.

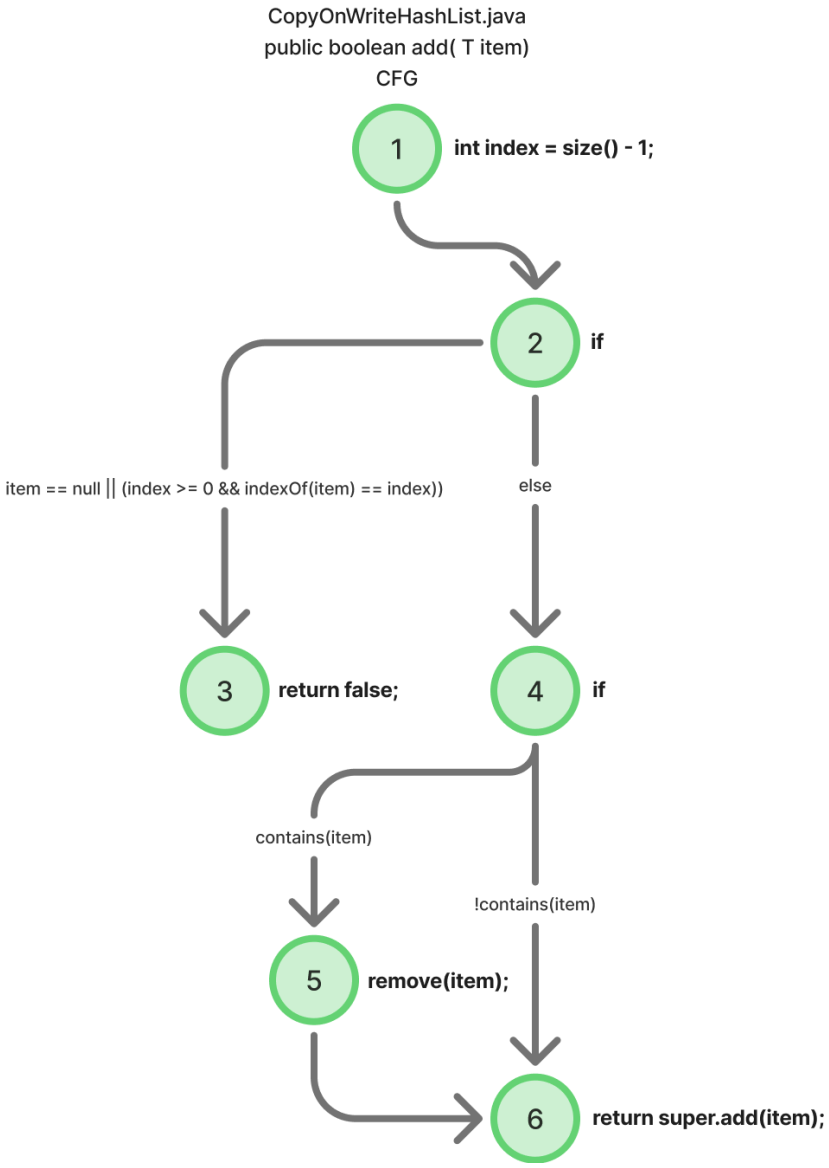

Figure 10. CFG designed for Figure 9

Node coverage was tested for this CFG where table 6 below illustrates the test requirements and test paths that will be implemented.

Table 6. Node Coverage analysis for Figure 10

Node Coverage		
Test Requirements	$TR(NC) = set\ of\ nodes\ in\ the\ graph$ $= \{ [1], [2], [3], [4], [5], [6] \}$	
Test Paths	$T1 = \{ 1, 2, 3 \}$	This test triggers the path where the input item is null.
	$T2 = \{ 1, 2, 4, 6 \}$	This test triggers the path where the item is new and not yet in the list.
	$T3 = \{ 1, 2, 4, 5, 6 \}$	This test triggers the path where the item already exists in the list.

Based on this table, the following JUnit test class seen in Figure11 was made to ensure node coverage. Its corresponding test results can be seen in Figure 12.

```java
package com.liskovsoft.smartyoutubetv2.common.utils;

import static org.junit.Assert.*;

public class CopyOnWriteHashListTest_CFG {

    /**
     * Path: T1 = {1, 2, 3}
     * Tests when item is null
     * Should return false and stop at node 3
     */
    @Test
    public void testPath1() {
        CopyOnWriteHashList<String> list = new CopyOnWriteHashList<>();
        boolean result = list.add(null);
        assertFalse(result); // Expect false since null items are rejected
    }

    /**
     * Path: T2 = {1, 2, 4, 6}
     * Tests when item is not in the list
     * Should skip removal and go to add directly
     */
    @Test
    public void testPath2() {
        CopyOnWriteHashList<String> list = new CopyOnWriteHashList<>();
        boolean result = list.add("newItem");
        assertTrue(result); // Expect true since it's a new item and gets added
    }

    /**
     * Path: T3 = {1, 2, 4, 5, 6}
     * Tests when the item is already in the list, but not at the last index
     * Should remove it first, then add it again
     */
    @Test
    public void testPath3() {
        CopyOnWriteHashList<String> list = new CopyOnWriteHashList<>();
        list.add("item");   // Add "item" at index 0
        list.add("item2");  // Add a second item to make "item" not the last element

        boolean result = list.add("item"); // Triggers remove() and re-add
        assertTrue(result);
    }
}
```

Figure 11. Test case implemented to satisfy node coverage for Figure 9

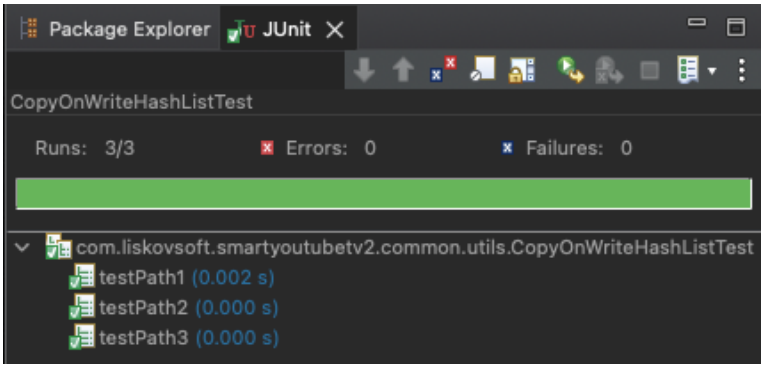

Figure 12. Test case results for Figure 9

For this method, the following DFG seen in Figure 13 was designed.

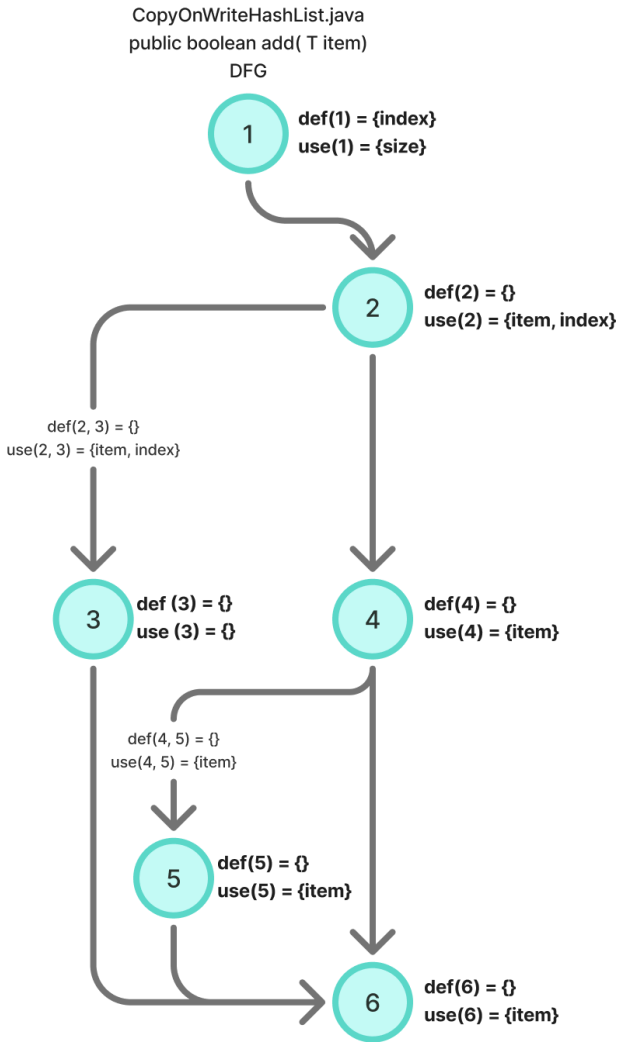

Figure 13. DFG designed for Figure 9

Use coverage was tested for this CFG where table 7 below illustrates the test requirements and test paths that will be implemented.

Table 7. Use Coverage analysis for Figure 13

Use Coverage			
Test Case	*Target Node*	*Variable Used*	*Purpose*
TC1	Node 2	index	Make sure index from size() is being used correctly
TC2	Node 2	item	Confirm that null items are rejected
TC3	Node 4	item	Test condition when item already exists
TC4	Node 5	item	Make sure item is removed when duplicated
TC5	Node 6	item	Check if item is added correctly to the list
TC6	Node 2	index, item	Test a case where item is at the end
TC7	Node 6	item	Confirm new items are handled correctly
TC8	Node 2	index	Cover edge case with an empty list

Based on this analysis, the following JUnit test class seen in Figure 14 was made to satisfy use coverage. Its corresponding test results can be seen in Figure 15.

```
1  package com.liskovsoft.smartyoutubetv2.common.utils;
2
3  import static org.junit.Assert.*;
4  import org.junit.Before;
5  import org.junit.Test;
6
7  public class CopyOnWriteHashListTest_DFG {
8
9      private CopyOnWriteHashList<String> list;
10
11     @Before
12     public void setUp() {
13         list = new CopyOnWriteHashList<>();
14     }
15
```

```java
    /*
     * TC1: Use of 'index' in condition (index >= 0 && indexOf(item) == index)
     * Target Node: Node 2
     */
    @Test
    public void testCase1() {
        list.add("A");                      // index = 0
        boolean result = list.add("A");  // indexOf(A) == index
        assertFalse(result);             // should return false
    }


    /*
     * TC2: Use of 'item' in null check
     * Target Node: Node 2
     */
    @Test
    public void testCase2() {
        boolean result = list.add(null);
        assertFalse(result);             // null should not be added
    }


    /*
     * TC3: Use of 'item' in contains(item) check
     * Target Node: Node 4
     */
    @Test
    public void testCase3() {
        list.add("B");
        list.add("X");                      // push B to index 0
        boolean result = list.add("B");  // B is not last anymore
        assertTrue(result);              // B should be re-added
    }
```

```java
    /*
     * TC4: Use of 'item' in remove(item)
     * Target Node: Node 5
     */
    @Test
    public void testCase4() {
        list.add("D");
        list.add("E");
        boolean result = list.add("D");  // triggers remove(D), then add(D)
        assertTrue(result);

        // Check that D appears only once and at the last position
        assertEquals("D", list.get(list.size() - 1));
        assertEquals(1, list.stream().filter(x -> x.equals("D")).count());
    }


    /*
     * TC5: Use of 'item' in super.add(item)
     * Target Node: Node 6
     */
    @Test
    public void testCase5() {
        boolean result = list.add("F");
        assertTrue(result);
        assertEquals("F", list.get(0));
    }


    /*
     * TC6: Use of 'index' and 'item' in full complex condition
     * Target Node: Node 2
     */
    @Test
    public void testCase6() {
        list.add("G");
        list.add("H");
        boolean result = list.add("G");   // index = 1, indexOf(G) = 0 → condition false
        assertTrue(result);               // allowed to add again (after removal)
    }
```

```
95      /*
96       * TC7: Add a brand-new item
97       * Target Node: Node 6
98       */
99      @Test
100     public void testCase7() {
101         boolean result = list.add("I");
102         assertTrue(result);              // item not in list, should be added
103     }
104
105
106     /*
107      * TC8: index = -1 (list is empty)
108      * Target Node: Node 2
109      */
110     @Test
111     public void testCase8() {
112         boolean result = list.add("J");
113         assertTrue(result);              // index = -1, skip first condition
114     }
115 }
```

Figure 14. Test case implemented to satisfy use coverage for Figure 9

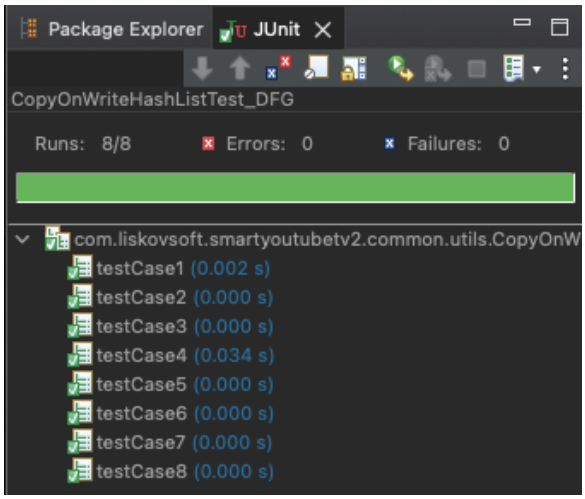

Figure 15. Test case results for Figure 9

Example #2:

The method *public boolean add(T item)* belonging to the java class *CopyOnWriteHashList.java* java was tested as seen in Figure 16.

```
public final boolean equals(Object obj) {
    if (obj == null || !(obj instanceof Proxy))
        return false;
    Proxy p = (Proxy) obj;
    if (p.type() == type()) {
        if (address() == null) {
            return (p.address() == null);
        } else
            return address().equals(p.address());
    }
    return false;
}
```

Figure 16. Method under test for graph based testing

For this method, the following CFG seen in Figure 17 was designed.

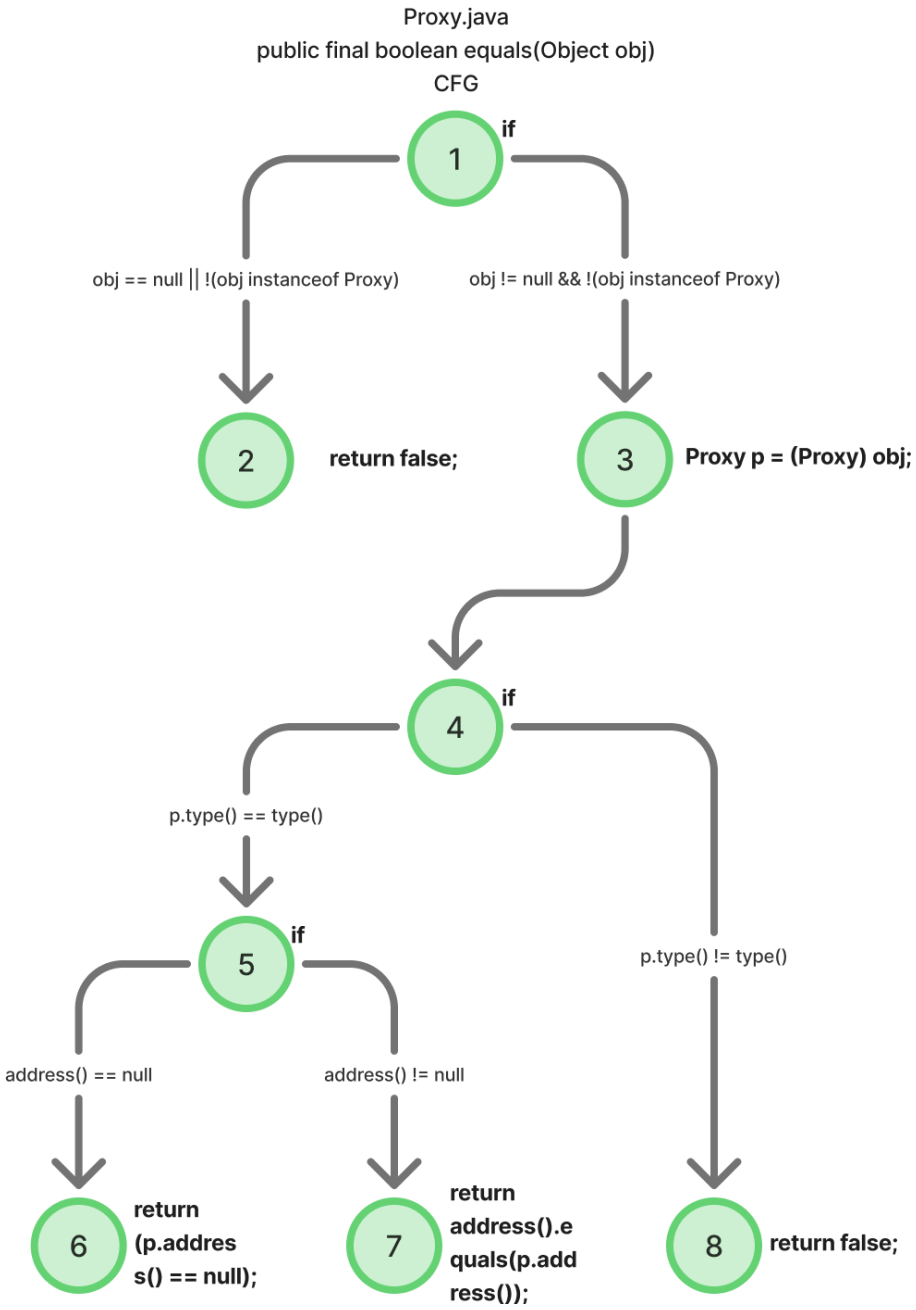

Figure 17. CFG designed for Figure 16

Node coverage was tested for this CFG where table 8 below illustrates the test requirements and test paths that will be implemented.

Table 8. Node Coverage analysis for Figure 17

Node Coverage		
Test Requirements	$TR(NC) = set\ of\ nodes\ in\ the\ graph$ $= \{ [1], [2], [3], [4], [5], [6], [7], [8] \}$	
Test Paths	$T1 = \{ 1, 2 \}$	This test checks when the input is null or not a Proxy.
	$T2 = \{ 1, 3, 4, 8 \}$	This test checks when the types don't match.
	$T3 = \{ 1, 3, 4, 5, 6 \}$	This test checks when both types match and addresses are null.
	$T4 = \{ 1, 3, 4, 5, 7\}$	This test checks when both types and addresses match.

Based on this table, the following JUnit test class seen in Figure 18 was made to ensure node coverage. Its corresponding test results can be seen in Figure 19.

```java
package com.liskovsoft.smartyoutubetv2.common.proxy;

import org.junit.Test;
import static org.junit.Assert.*;
import java.net.SocketAddress;

public class ProxyTest_CFG {

    // Helper method
    private static PasswdInetSocketAddress createAddress(String host, int port, String user, String pass) {
        return PasswdInetSocketAddress.createUnresolved(host, port, user, pass);
    }

    /**
     * Path: T1 = {1, 2}
     * Tests when obj is null or not a Proxy
     * Should return false at node 2
     */
    @Test
    public void testPath1() {
        Proxy proxy = Proxy.NO_PROXY;

        assertFalse(proxy.equals(null));          // null test
        assertFalse(proxy.equals("NotAProxy"));   // type mismatch test
    }

    /**
     * Path: T2 = {1, 3, 4, 8}
     * Tests when obj is a Proxy but type does not match
     * Should return false at node 8
     */
    @Test
    public void testPath2() {
        SocketAddress addr = createAddress("localhost", 8080, "user", "pass");
        Proxy proxy1 = new Proxy(Proxy.Type.HTTP, addr);
        Proxy proxy2 = new Proxy(Proxy.Type.SOCKS, addr);

        assertFalse(proxy1.equals(proxy2)); // type mismatch
    }

```

```java
    /**
     * Path: T3 = {1, 3, 4, 5, 6}
     * Tests when both proxies are NO_PROXY with null addresses
     * Should return true at node 6
     */
    @Test
    public void testPath3() {
        Proxy proxy1 = Proxy.NO_PROXY;
        Proxy proxy2 = Proxy.NO_PROXY;

        assertTrue(proxy1.equals(proxy2)); // both type == DIRECT, address == null
    }

    /**
     * Path: T4 = {1, 3, 4, 5, 7}
     * Tests when types and addresses match
     * Should return true at node 7
     */
    @Test
    public void testPath4() {
        SocketAddress addr = createAddress("localhost", 8080, "user", "pass");
        Proxy proxy1 = new Proxy(Proxy.Type.HTTP, addr);
        Proxy proxy2 = new Proxy(Proxy.Type.HTTP, addr);

        assertTrue(proxy1.equals(proxy2)); // type + address match
    }

    /**
     * Path: T5 = {1, 3, 4, 5, 7}
     * Tests when types match but addresses differ
     * Should return false at node 7
     */
    @Test
    public void testPath5() {
        SocketAddress addr1 = createAddress("localhost", 8080, "user1", "pass1");
        SocketAddress addr2 = createAddress("localhost", 8080, "user2", "pass2");

        Proxy proxy1 = new Proxy(Proxy.Type.HTTP, addr1);
        Proxy proxy2 = new Proxy(Proxy.Type.HTTP, addr2);

        assertFalse(proxy1.equals(proxy2)); // same type, different address
    }
}
```

Figure 18. Test case implemented to satisfy node coverage for Figure 16

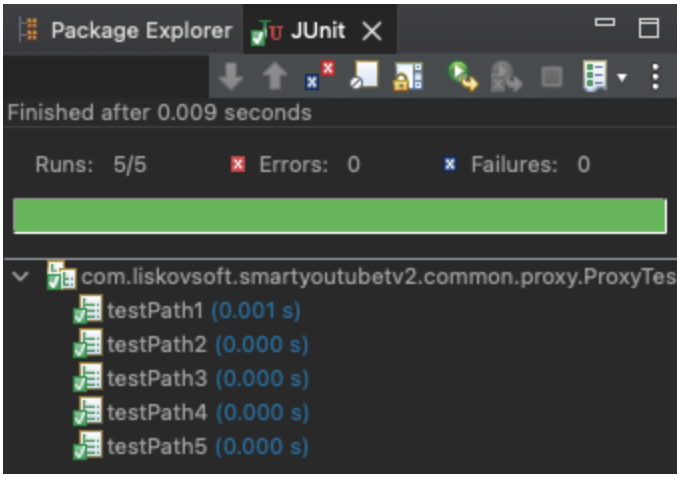

Figure 19. Test case results for Figure 16

For this method, the following DFG seen in Figure 20 was designed.

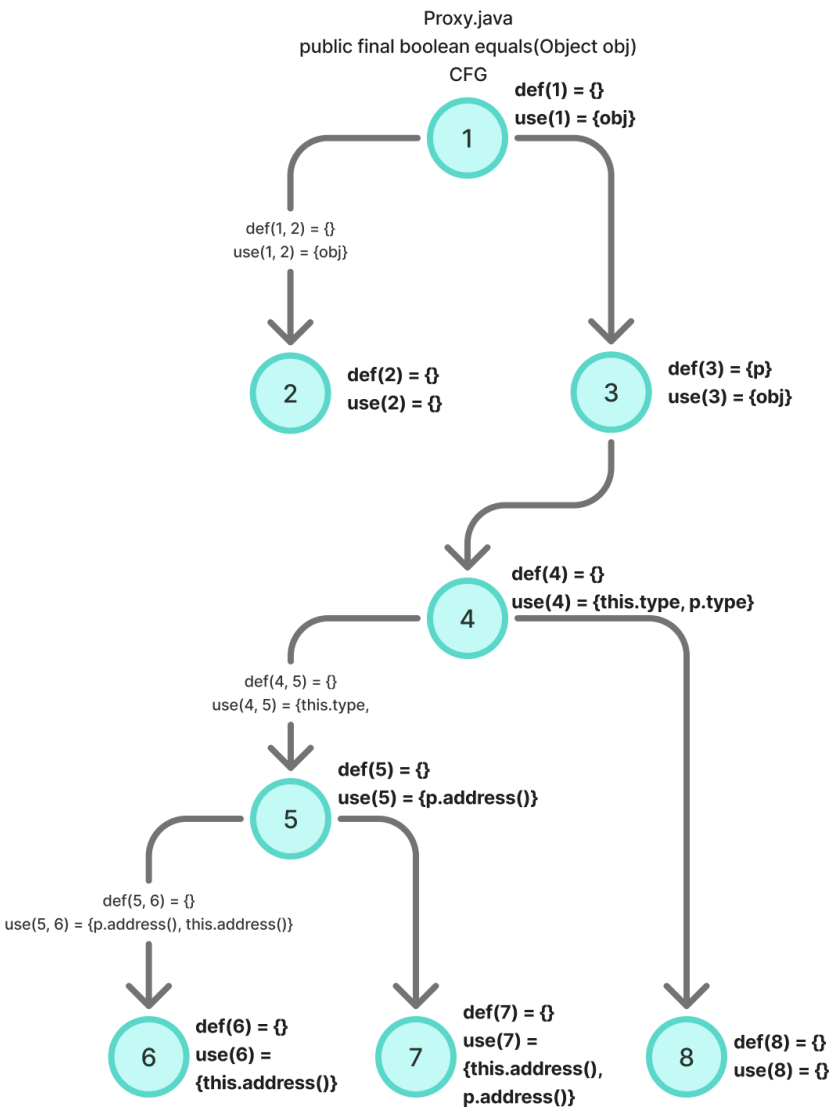

Figure 20. DFG designed for Figure 16

Use coverage was tested for this CFG where table 9 below illustrates the test requirements and test paths that will be implemented.

Table 9. Use Coverage analysis for Figure 20

Use Coverage			
Test Case	*Target Node*	*Variable Used*	*Purpose*
TC1	Node 2	obj	Check if obj is null or not a Proxy
TC2	Node 3	obj	Make sure casting obj to Proxy works

TC3	Node 4	p.type, this.type	Test if both proxies have the same type
TC4	Node 5	p.address()	Check if p.address() is null
TC5	Node 6	this.address()	Confirm this.address() is null
TC6	Node 7	this.address(), p.address()	Test when both addresses are not null and equal
TC7	Node 8	this.address(), p.address()	Test when both addresses are not equal

Based on this analysis, the following JUnit test class seen in Figure 21 was made to satisfy use coverage. Its corresponding test results can be seen in Figure 22.

```java
package com.liskovsoft.smartyoutubetv2.common.proxy;

import org.junit.Test;
import java.net.SocketAddress;
import static org.junit.Assert.*;

public class ProxyTest_DFG {

    // Helper method
    private PasswdInetSocketAddress addr(String host, int port) {
        return PasswdInetSocketAddress.createUnresolved(host, port, "user", "pass");
    }

    /* TC1: Check if obj is null or not a Proxy
     * Target Node: Node 2
     */
    @Test
    public void testCase1() {
        Proxy proxy = new Proxy(Proxy.Type.HTTP, addr("localhost", 8080));
        boolean result = proxy.equals(null);
        assertFalse(result);
    }

    /* TC2: Make sure casting obj to Proxy works
     * Target Node: Node 3
     */
    @Test
    public void testCase2() {
        Proxy proxy = new Proxy(Proxy.Type.HTTP, addr("localhost", 8080));
        boolean result = proxy.equals("not a proxy");
        assertFalse(result);
    }

    /* TC3: Test if both proxies have the same type
     * Target Node: Node 4
     */
    @Test
    public void testCase3() {
        Proxy proxy1 = Proxy.NO_PROXY;
        Proxy proxy2 = Proxy.NO_PROXY;
        boolean result = proxy1.equals(proxy2);
        assertTrue(result);
    }

```

```java
    /* TC4: Check if p.address() is null
     * Target Node: Node 5
     */
    @Test
    public void testCase4() {
        Proxy proxy1 = Proxy.NO_PROXY;
        Proxy proxy2 = new Proxy(Proxy.Type.HTTP, addr("localhost", 8080));
        boolean result = proxy1.equals(proxy2);
        assertFalse(result);
    }

    /* TC5: Confirm this.address() is null
     * Target Node: Node 6
     */
    @Test
    public void testCase5() {
        Proxy proxy1 = Proxy.NO_PROXY; // this.address() = null
        Proxy proxy2 = new Proxy(Proxy.Type.HTTP, addr("localhost", 8080));
        boolean result = proxy1.equals(proxy2);
        assertFalse(result); /* should return false as address() == null */
    }


    /* TC6: Test when both addresses are not null and equal
     * Target Node: Node 7
     */
    @Test
    public void testCase6() {
        SocketAddress shared = addr("localhost", 8080);
        Proxy proxy1 = new Proxy(Proxy.Type.HTTP, shared);
        Proxy proxy2 = new Proxy(Proxy.Type.HTTP, shared);
        boolean result = proxy1.equals(proxy2);
        assertTrue(result);
    }

    /* TC7: Test when both addresses are not equal
     * Target Node: Node 8
     */
    @Test
    public void testCase7() {
        Proxy proxy1 = new Proxy(Proxy.Type.HTTP, addr("localhost", 8080));
        Proxy proxy2 = new Proxy(Proxy.Type.HTTP, addr("127.0.0.1", 8080));
        boolean result = proxy1.equals(proxy2);
        assertFalse(result);
    }

}
```

Figure 21. Test case implemented to satisfy use coverage for Figure 16

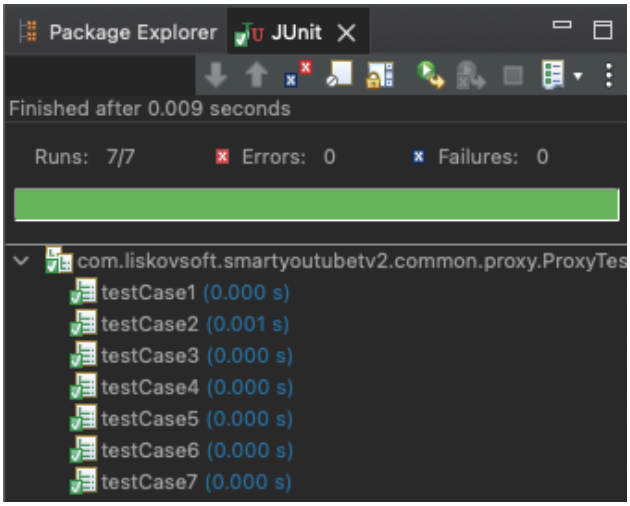

Figure 22. Test case results for Figure 16

Test Approach #3 - Logic-Based Testing:

Example #1:

Class 1: CopyOnWriteHashList.java
Method1: Test Logic for : public boolean add(T item)

This method is here to ensure that duplicate or null items are not added to the list. It first checks whether the item is null or already exists at the last index of the list. If either of these conditions is true, the method returns false and does not add the item. If the item is found anywhere else in the list, it is removed before being re-added to the end. This logic is going to help maintain a unique, updated order of items in the list. For testing purposes, we focus on the condition if (item == null || (index >= 0 && indexOf(item) == index)), which controls whether the method proceeds or exits early. We simplify this logic to two conditions: C1 = item == null and C2 = item is already at the end. The decision is then represented as D = !(C1 OR C2), which we test using logic coverage techniques like GACC, CACC, and RACC.

Testing the decision for code:
if (item == null || (index >= 0 && indexOf(item) == index))

Simplify it to the logic condition:
C1 = item == null
C2 = item is already at the end of the list
Decision: D = !(C1 OR C2)

Truth Table:

Row#	c1	c2	P	Pc1	Pc2
1	T	T			
2	T			T	
3		T			T
4			T	T	T

Figure 23. Truth Table

The following result for GACC is based on the truth table on the right:

Major Clause	Set of possible tests
c1	(2,4)
c2	(3,4)

Figure 24. GACC

The following result for CACC is based on the truth table on the right:

Major Clause	Set of possible tests
c1	(2,4)
c2	(3,4)

Figure 25. CACC

The following result for RACC is based on the truth table on the right:

Major Clause	Set of possible tests
c1	(2,4)
c2	(3,4)

Figure 26. RACC

Example #2:

Method 2: public void add(int index, T item)

This method is here to add an item at a specific index in the list, but only if the item is valid and not already positioned at the given index. The logic is going to ensure that null items or duplicates at the same position are not reinserted. Specifically, the method checks whether the item is null, or if the item already exists at the target index regardless in both cases, the insertion is skipped. Otherwise, if the item exists elsewhere in the list, it is first removed and then added either at the given index (if valid) or appended at the end. For testing purposes, we focus on the condition if (item == null || (index >= 0 && indexOf(item) == index)), which determines whether the method exits early. We break this down into three conditions: C1 = item == null, C2 = index >= 0, and C3 = indexOf(item) == index. The final decision is expressed as D = !(C1 OR (C2 AND C3)), and is tested using truth tables and logic-based coverage methods.

Testing the Decision for Code:
if (item == null || (index >= 0 && indexOf(item) == index))

Simplify it to the Logic Condition:
C1 = item == null
C2 = index >= 0
C3 = indexOf(item) == index
Decision: D = !(C1 OR (C2 AND C3))

Truth Table:

Row#	C1	C2	C3	P	PC1	PC2	PC3
1	T	T	T				
2	T	T			T		
3	T		T		T		
4	T				T		
5		T	T			T	T
6		T		T	T		T
7			T	T	T	T	
8				T	T		

Figure 27. Truth Table

The following result for GACC is based on the truth table on the right:

Major Clause	Set of possible tests
C1	(2,6), (2,7), (2,8), (3,6), (3,7), (3,8), (4,6), (4,7), (4,8)
C2	(5,7)
C3	(5,6)

Figure 28. GACC

The following result for CACC is based on the truth table on the right:

Major Clause	Set of possible tests
C1	(2,6), (2,7), (2,8), (3,6), (3,7), (3,8), (4,6), (4,7), (4,8)
C2	(5,7)
C3	(5,6)

Figure 29. CACC

The following result for RACC is based on the truth table on the right:

Major Clause	Set of possible tests
C1	(2,6), (3,7), (4,8)
C2	(5,7)
C3	(5,6)

Figure 30. RACC

Test Approach #4 - Mutation Testing:

Mutation is a form of white box testing where a tester knows the code to be tested. The code is modified in specific ways to create mutations, and what determines how mutations are formed are called mutation operators. Mutation testing involves mutating code to see if errors will occur. When a mutation causes an error in the code, it is considered that that mutation is killed. Mutations that do not cause an error are considered to have survived. Ideally mutation unit testing should result in a high percentage of killed mutations, or a mutation score approaching 100%. This would mean that the code under test is robust and highly responsive to any misuse.

Example #1:

Class: CopyOnWriteHashList.java
Methods: public boolean add(T item) & public void add(int index, T item)

```
public class CopyOnWriteHashList<T> extends CopyOnWriteArrayList<T> {
    @Override
    public boolean add(T item) {
        int index = size() - 1;
        if (item == null || (index >= 0 && indexOf(item) == index)) {
            return false;
        } else if (contains(item)) {
            remove(item);
        }

        return super.add(item);
    }

    @Override
    public void add(int index, T item) {
        if (item == null || (index >= 0 && indexOf(item) == index)) {
            return;
        } else if (contains(item)) {
            remove(item);
        }

        if (index >= 0 && index < size()) {
            super.add(index, item);
        } else {
            super.add(item);
        }
    }
}
```

Figure 31. public boolean add(T item) & public void add(int index, T item)

Test:

```
public class CopyOnWriteHashListTest {
    private CopyOnWriteHashList<String> list;

    @Before
    public void setUp() {
        list = new CopyOnWriteHashList<>();
    }

    @Test
    public void testAdd_nullItem_shouldNotBeAdded() {
        boolean result = list.add(null);
        assertFalse(result);
        assertTrue(list.isEmpty());
    }

    @Test
    public void testAdd_uniqueItem_shouldBeAdded() {
        boolean result = list.add("Apple");
        assertTrue(result);
        assertEquals(1, list.size());
        assertEquals("Apple", list.get(0));
    }

    @Test
    public void testAdd_duplicateAtEnd_shouldNotAddAgain() {
        list.add("Banana");
        boolean result = list.add("Banana"); // should be ignored because it's at end
        assertFalse(result);
        assertEquals(1, list.size());
    }
```

```
    @Test
    public void testAdd_duplicateNotAtEnd_shouldBeMovedToEnd() {
        list.add("A");
        list.add("B");
        list.add("C");

        boolean result = list.add("B"); // should move B to end
        assertTrue(result);
        assertEquals(3, list.size());
        assertEquals("C", list.get(1));
        assertEquals("B", list.get(2)); // B now at end
    }

    @Test
    public void testAddIndex_validIndex_uniqueItem_shouldAddAtIndex() {
        list.add("X");
        list.add("Y");

        list.add(1, "Z");

        assertEquals(3, list.size());
        assertEquals("X", list.get(0));
        assertEquals("Z", list.get(1));
        assertEquals("Y", list.get(2));
    }
```

```
    @Test
    public void testAddIndex_invalidIndex_shouldAppend() {
        list.add("One");
        list.add(100, "Two"); // out of bounds, so should append

        assertEquals(2, list.size());
        assertEquals("Two", list.get(1));
    }

    @Test
    public void testAddIndex_duplicateAtSameIndex_shouldNotAddAgain() {
        list.add("First");
        list.add(0, "First"); // same index

        assertEquals(1, list.size());
        assertEquals("First", list.get(0));
    }

    @Test
    public void testAddIndex_duplicateNotAtIndex_shouldMoveToIndex() {
        list.add("One");
        list.add("Two");
        list.add(0, "Two"); // remove and insert at 0

        assertEquals(2, list.size());
        assertEquals("Two", list.get(0));
        assertEquals("One", list.get(1));
    }
}
```

Table 10. Test Cases for public boolean add(T item) & public void add(int index, T item)

Pit Test Coverage Report

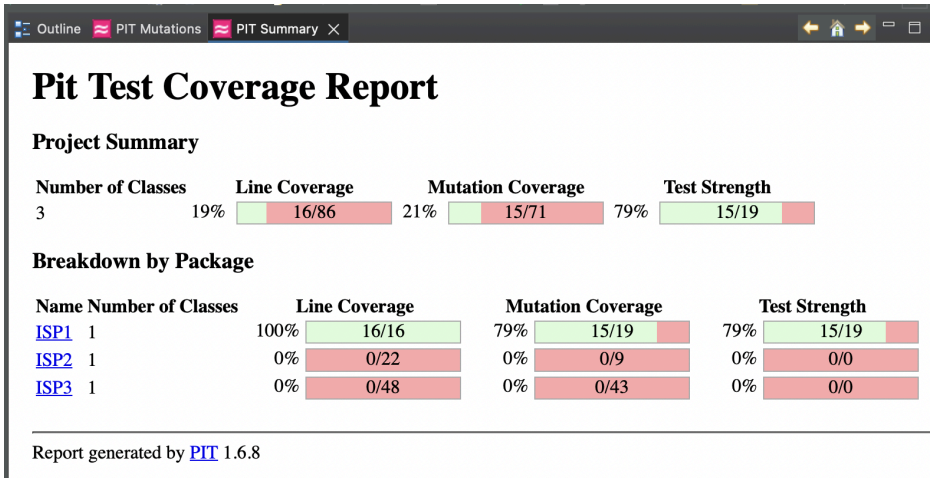

Figure 32. PIT Results for public boolean add(T item) & public void add(int index, T item)

The mutation testing results indicate a strong level of test effectiveness. As shown in the report, 79% of mutations were killed, which suggests that our unit tests were able to catch the majority of faults. Additionally, we achieved 100% line coverage for the method under test. This means that every line of code was executed during testing, demonstrating thorough code coverage. However, it's important to note that line coverage alone doesn't guarantee high test quality, which is why mutation testing is so valuable, it evaluates whether the tests are truly meaningful by checking if they can detect small, deliberate changes (mutants) in the code. The fact that some mutants survived highlights areas where the current tests could be improved to better assert specific behaviors or edge cases.

Pit Mutation Summary

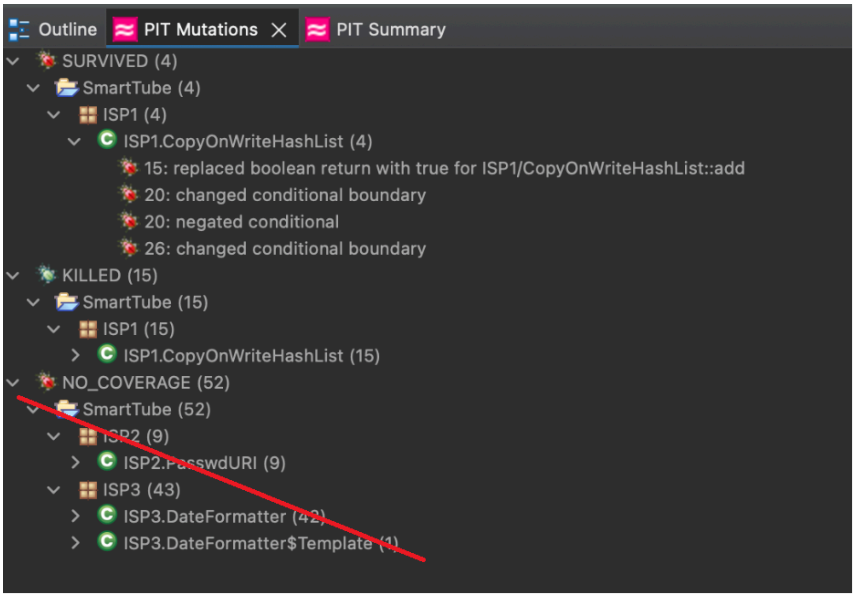

Figure 33. PIT Summary for public boolean add(T item) & public void add(int index, T item)

Out of a total of 19 mutants, 15 were killed by our test suite, and only 4 survived. There were 0 mutants with no coverage, meaning all mutations were at least executed during the test run. A 79% mutation kill rate demonstrates a strong foundation of test quality, particularly given the complexity of certain logic within the method. However, the 4 surviving mutants suggest there may be specific conditions or edge cases that our current tests didn't fully address. Investigating these surviving mutations can highlight areas for improvement, such as enhancing existing tests or adding new ones to target those overlooked paths more precisely.

Selected Mutation Operators

- CONDITIONALS_BOUNDARY
- NEGATE_CONDITIONALS
- MATH
- RETURN_VALS
- REMOVE_CONDITIONALS

Example #2:

Class: PasswdURI.java
Methods: public String getUsername() & public String getPassword()

```
public String getUsername() {
    String authority = mURI.getAuthority();
    String[] split = authority.split("@");

    String result = null;

    if (split.length == 2) {
        String[] split2 = split[0].split(":");

        if (split2.length == 2) {
            result = split2[0];
        }
    }

    return result;
}

public String getPassword() {
    String authority = mURI.getAuthority();
    String[] split = authority.split("@");

    String result = null;

    if (split.length == 2) {
        String[] split2 = split[0].split(":");

        if (split2.length == 2) {
            result = split2[1];
        }
    }

    return result;
}
```

Figure 34. public String getUsername() & public String getPassword()

Test:

```java
public class PasswdURITest {

    @Test
    public void testGetUsernameAndPassword_normal() throws URISyntaxException {
        PasswdURI uri = new PasswdURI("http://user123:pass123@127.0.0.1:8080");

        assertEquals("user123", uri.getUsername());
        assertEquals("pass123", uri.getPassword());
    }

    @Test
    public void testGetUsernameAndPassword_missingColon() throws URISyntaxException {
        PasswdURI uri = new PasswdURI("http://user123@127.0.0.1:8080");

        assertNull(uri.getUsername());
        assertNull(uri.getPassword());
    }

    @Test
    public void testGetUsernameAndPassword_missingAtSymbol() throws URISyntaxException {
        PasswdURI uri = new PasswdURI("http://127.0.0.1:8080");

        assertNull(uri.getUsername());
        assertNull(uri.getPassword());
    }

    @Test
    public void testGetUsernameAndPassword_colonButNoPassword() throws URISyntaxException {
        PasswdURI uri = new PasswdURI("http://user123:@127.0.0.1:8080");

        assertEquals("user123", uri.getUsername());
        assertEquals("", uri.getPassword()); // still valid: user: empty password
    }

    @Test
    public void testGetUsernameAndPassword_emptyAuthority() throws URISyntaxException {
        PasswdURI uri = new PasswdURI("http://@127.0.0.1:8080");

        assertNull(uri.getUsername());
        assertNull(uri.getPassword());
    }

    @Test
    public void testGetUsernameAndPassword_extraColonsInCredentials() throws URISyntaxException
        PasswdURI uri = new PasswdURI("http://user:extra:pass@127.0.0.1:8080");

        // split2.length != 2 → should be null
        assertNull(uri.getUsername());
        assertNull(uri.getPassword());
    }
}
```

Table 11. Test Cases for public String getUsername() & public String getPassword()

Pit Test Coverage Report

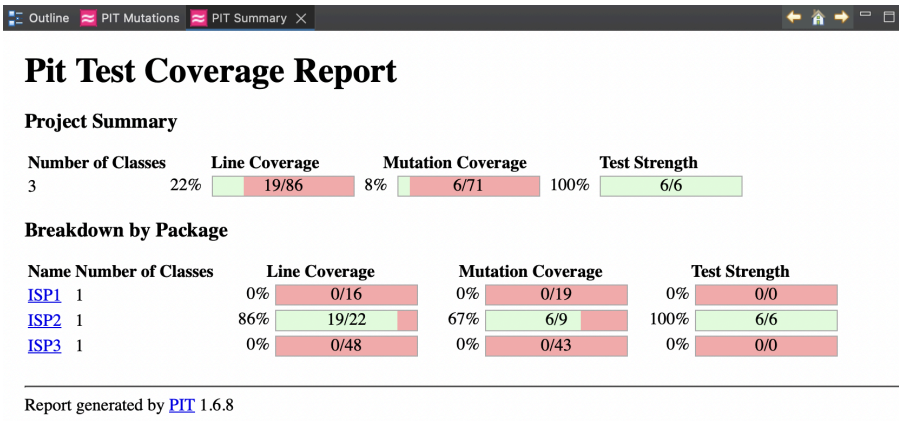

Figure 35. PIT Results for public String getUsername() & public String getPassword()

The PIT test coverage report reflects a strong level of confidence in the unit tests written for the targeted method with 67% of mutants being killed and 100% line coverage. Although the overall class coverage is reported at 86%, it's important to clarify that this percentage is influenced by additional methods and logic within the class that were not within the intended scope of testing. These untested portions were either unrelated or outside the focus of the current test objectives. For the specific method being evaluated, 100% line coverage was successfully achieved, meaning every line of executable code was reached and assessed during testing. This level of coverage suggests that the test cases are thorough in terms of control flow and execution, ensuring that the method behaves as expected under various input scenarios.

Pit Mutation Summary

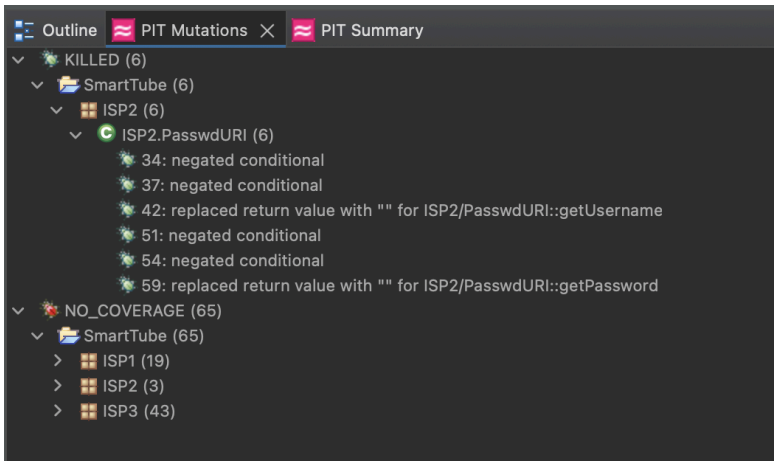

Figure 36. PIT Summary for public String getUsername() & public String getPassword()

The mutation testing results display to us that 6 out of 9 mutants were killed, giving a 67% mutation kill rate. This value indicates that a significant portion of potential faults—introduced by small code changes (mutants), were effectively detected and rejected by the test suite. However, 3 mutants survived, all due to lack of coverage, meaning that those specific branches or conditions in the code were not exercised at all by the tests. Overall, the results validate the strength of the current tests while also having the option to improve the test cases in the future to reach full mutation coverage.

Selected Mutation Operators

- CONDITIONALS_BOUNDARY
- NEGATE_CONDITIONALS
- MATH
- RETURN_VALS
- REMOVE_CONDITIONALS

Findings and Results

The ISP testing revealed a fault in the Proxy.java class, in the constructor *public Proxy(Type type, SocketAddress sa)*. When examining the code for the constructor, Figure 37, it can be seen that the constructor has no null handling for the type. The if statement only checks if the type is DIRECT; passing type = null, as seen in Figure 38, will render the condition as false. A valid sa is passed, rendering the second condition of the if statement as false as well, therefore the *IllegalArgumentException* will not be thrown and the Proxy object with a null type should be successfully created as seen in the test results above (Figure 8). This fault could prove to be an issue as an incorrect connection may be instantiated, causing issues in later parts of the code/process.

```
/**
 * Creates an entry representing a PROXY connection.
 * Certain combinations are illegal. For instance, for types Http, and
 * Socks, a SocketAddress <b>must</b> be provided.
 * <P>
 * Use the {@code Proxy.NO_PROXY} constant
 * for representing a direct connection.
 *
 * @param type the {@code Type} of the proxy
 * @param sa the {@code SocketAddress} for that proxy
 * @throws IllegalArgumentException when the type and the address are
 * incompatible
 */
public Proxy(Type type, SocketAddress sa) {
    if ((type == Type.DIRECT) || !(sa instanceof PasswdInetSocketAddress))
        throw new IllegalArgumentException("type " + type + " is not compatible with address " + sa);
    this.type = type;
    this.sa = sa;
}
```

Figure 37. Constructor method Proxy and its description.

```
/**
 * Test case: Passing null as Proxy.Type should still construct Proxy
 * since there is no null check and condition evaluates safely.
 */
@Test
public void testProxyConstructor_nullTypeAllowed() {
    SocketAddress sa = PasswdInetSocketAddress.createUnresolved("localhost", 8080, "user", "pass");
    Proxy proxy = new Proxy(null, sa);
    assertNull(proxy.type());
    assertEquals(sa, proxy.address());
}
```

Figure 38. Null type Proxy constructor method test.

The ISP testing revealed a fault in the PasswdURI class when analyzing its handling of malformed or incomplete URIs. As seen in the test cases above, particularly testMalformedURIMissingScheme and testMalformedURIMissingHost, the constructor fails to correctly parse certain user inputs that lack a proper scheme or host, leading to unexpected behavior or exceptions. For example, a URI string like "user:pass@proxy.example.com:8080" is accepted as a valid input by standard URI parsers but fails in PasswdURI due to its reliance on explicit scheme parsing. This exposes a potential weakness in the implementation, where user credentials may appear valid syntactically but do

not get parsed correctly or are rejected altogether. Such faults can result in URI-based authentication or routing logic breaking down later in execution, ultimately compromising reliability or security in systems that depend on precise URI parsing.

```
@Test(expected = URISyntaxException.class)
public void testMalformedURIMissingScheme() throws URISyntaxException {
   new PasswdURI("user:pass@proxy.example.com:8080");
}

@Test(expected = URISyntaxException.class)
public void testMalformedURIMissingHost() throws URISyntaxException {
   new PasswdURI("http://user:pass@:8080");
}
```

Figure 39. testMalformedURIMissingScheme() and Host() constructor method test.

Reflections and Conclusions

One of the main challenges encountered during testing was the complex set of dependencies and external imports used across various Java classes in the SmartTube application. SmartTube is a multi-language Android-based project that relies on the Maven build system, making integration into the Eclipse Java IDE difficult. Many classes could not be imported successfully due to unresolved dependencies, making it infeasible to test those classes and their methods in the current environment.

Additionally, since the application is built for Android, and the Eclipse IDE is not typically used for Android development, the team faced limitations due to their unfamiliarity with Android-specific tools, like Mockito, and Maven. This, combined with limited experience beyond the JUnit testing framework, restricted the range of methods that could be effectively tested. Going forward, it is clear that setting up the appropriate development and testing environments and gaining familiarity with supporting tools is essential when testing modern Android applications.

Despite these constraints, the testing that was completed proved valuable. Several issues were uncovered even within the relatively small subset of methods examined. In total, ten methods across five classes were tested. To give context, the SmartTube repository contains approximately twelve thousand files, many of which are Java classes, highlighting the sheer size and complexity of the codebase.

In conclusion, this project provided valuable insight into the fundamentals and importance of software testing, particularly unit testing. Applying the four testing strategies explored in the course: Input Space Partitioning (ISP), Control Flow and Data Flow Graphs (CFG/DFG), Logic-Based Testing, and Mutation Testing helped reinforce the value of creating thorough and effective test cases. Each approach

contributed to identifying potential issues and strengthening the reliability of the methods tested, demonstrating the real-world impact of thoughtful and systematic software testing.

References

[1] yuliskov, “GitHub - yuliskov/SmartTube: Advanced player for set-top boxes and tvs running Android OS,” *GitHub*, Mar. 04, 2025. https://github.com/yuliskov/SmartTube (accessed Mar. 9, 2025).