System Test Plan
Mutation Based Testing

By: Cail Keeling

Table of Contents

1. Mutants for Fancy String

1.1. The Problem

A fancy string is a string where no three consecutive characters are equal. Given a string s, delete the minimum possible number of characters from s to make it fancy. Return the final string after the deletion.

For example, given $aaabaaaa$ the output should be $aabaa$.

1.2. The Mutants

1.2.1. Mutant 1 - line 1 `StringBuilder sb = new StringBuilder();`

Description of mutant:

- **Mutation operator:** ASR
- **Change to source code:** StringBuilder sb = new StringBuilder("a");

This mutant is **strongly killable** based on the following:

- **Reachability:** True – it's the first line of code in the method
- **Infection:** True – assigning a value to a string builder always creates a different state than terminating the program.
- **Propagation:** True – the program terminating with no return is a different output than a string.

1.2.2. Mutant 2 - line 2 `for (int i = 0; i < s.length(); i++) {`

Description of mutant:

- **Mutation operator:** COR
- **Change to source code:** i < 1

This mutant is **strongly killable** based on the following:

- **Reachability:** True – there are no conditions prior to this line to trigger an error or a return
- **Infection:** True – the mutant will only run once and will only capture the first char of string s, infection occurs if s.length()>1
- **Propagation:** True – there is no way for the code to replace or remove different sb appendings.

1.2.3. Mutant 3 - line 3 `char thisChar = s.charAt(i);`

Description of mutant:

- **Mutation operator:** ASR
- **Change to source code:** char thisChar = s.charAt(i+1);

This mutant is **strongly killable** based on the following:

- **Reachability:** True – as long as s.length() >= 1, there is no code to result in an error or return before this line is read.
- **Infection:** True – whenever s.charAt(i) != s.charAt(i+1)
- **Propagation:** True – the first char appended to sb will be immediately wrong and therefore sb will be wrong

1.2.4. Mutant 4 - line 4 if **(i == 0)**

Description of mutant:

- **Mutation operator:** ASR
- **Change to source code:** if(i != 0)

This mutant is **strongly killable** based on the following:

- **Reachability:** True - as long as s.length() >= 1, there is no code to result in an error or return before this line is read.
- **Infection:** True – the first character of input string s will not be appended when in the original, it will be. There are scenarios where the first character getting skipped won't cause an infection (strings with three repeating letters in the front), but an infection is guaranteed later in the code when a third congruent character in a row is added without the proper venting
- **Propagation:** True – because of the absence of the first sb character and the presence of every s character in sb without any filtering, a different output is guaranteed

1.2.5. Mutant 5 - line 5 **sb.append(thisChar);**

Description of mutant:

- **Mutation operator:** ASR – because append is essentially sb += char
- **Change to source code:** sb.append(s.charAt(i+1));

This mutant is **strongly killable** based on the following:

- **Reachability:** True - as long as s.length() >= 1, there is no code to result in an error or return before this line is read. i starts at 0 at the beginning of the loop so there is no way for the loop to break before this line is read.
- **Infection:** True – whenever s.charAt(i) != s.charAt(i +1) whenever i = 0.
- **Propagation:** True – because the incorrect char is appended onto sb, the infection is guaranteed to make it to the returned string (there is no way in the program to replace the missing character in the correct spot)

1.2.6. Mutant 6 - line 6 else

Description of mutant:

- **Mutation operator:** COR
- **Change to source code:** else if(i == 1)

This mutant is **strongly killable** based on the following:

- **Reachability:** True – as long as s.length() >= 1, there is no code to result in an error or return before this line is read.
- **Infection:** if s.length() >= 2, then the 3rd char in s and any char afterwards will not be evaluated and added to sb. This will only not matter when string s has 3+ characters that are all the same.
- **Propagation:** True – because sb will be missing appended chars, and there is no way to add them back, the infection will always lead to a different output

1.2.7. Mutant 7 - line 7 char **lastChar = sb.charAt(sb.length() - 1);**

Description of mutant:

- **Mutation operator:** AOR
- **Change to source code:** char lastChar = sb.charAt(sb.length())

This mutant is **strongly killable** based on the following:

- **Reachability:** True – as long as s.length() >= 1, there is no code to result in an error or return before this line is read.
- **Infection:** True – an error will ensue whenever a char is grabbed at position length() bc position length()-1 is the end of the string. This will always be different than the original code.
- **Propagation:** True – the release of an error message gives the program no room to fix sb to release congruent output with the original.

1.2.8. Mutant 8 - line 8 char **lastLastChar = sb.length() >= 2 ? sb.charAt(sb.length() - 2) : ' ';**

Description of mutant:

- **Mutation operator:** COR
- **Change to source code:** char lastLastChar = sb.length() <= 2 ? sb.charAt(sb.length() - 2) : ‘ ‘;

This mutant is **strongly killable** based on the following:

- **Reachability:** True – as long as s.length() >= 1, there is no code to result in an error or return before this line is read.
- **Infection:** True – every time because an IndexOutOfBounds error will ensue
- **Propagation:** True – there is no way for the code to replace or remove different sb appendings.

1.2.9. Mutant 9 - line 9 if **(lastChar != thisChar && lastLastChar != thisChar) {**

Description of mutant:

- **Mutation operator:** COR
- **Change to source code:** if (lastChar != thisChar || lastLastChar != thisChar) {

This mutant is **strongly killable** based on the following:

- **Reachability:** True – as long as s.length() >= 1, there is no code to result in an error or return before this line is read.
- **Infection:** True – whenever a character in s is preceded by only 2 congruent characters before it, it won't be added to sb. In origin, the current char would be added only if both chars in sb weren't equal to the current char
- **Propagation:** True – because there is no way to remove or replace incorrectly appended chars in sb, the infection will always cause a different output.

1.2.10. Mutant 10 - line 10 `sb.append(thisChar);`

Description of mutant:

- **Mutation operator:** ASR – because append is essentially sb += char
- **Change to source code:** sb.append(lastChar);

This mutant is **strongly killable** based on the following:

- **Reachability:** True – as long as s.length() >= 1 and there is at least one character that is not preceded by 2 of the same chars as itself, there is no code to result in an error or return before this line is read.
- **Infection:** True – whenever this line is reached, it is already confirmed that lastChar != thisChar, therefore, replacing thisChar with lastChar will always result in an infection
- **Propagation:** True – because there is no way to remove or replace incorrectly appended chars in sb, the infection will always cause a different output.

1.2.11. Mutant 11 - line 11 return **sb.toString();**

Description of mutant:

- **Mutation operator:** BSR
- **Change to source code:** return “”; I am using an intentional blank return instead of creating a “Bomb()” function because I could not find any good literature on it.

This mutant is **strongly killable** based on the following:

- **Reachability:** True – unless an error is returned that terminates the program, this line will be reached.
- **Infection:** True – a termination of a program will always be different that the conversion of a StringBuilder to a string
- **Propagation:** True – the termination of a program will always result in a different output than returning a string

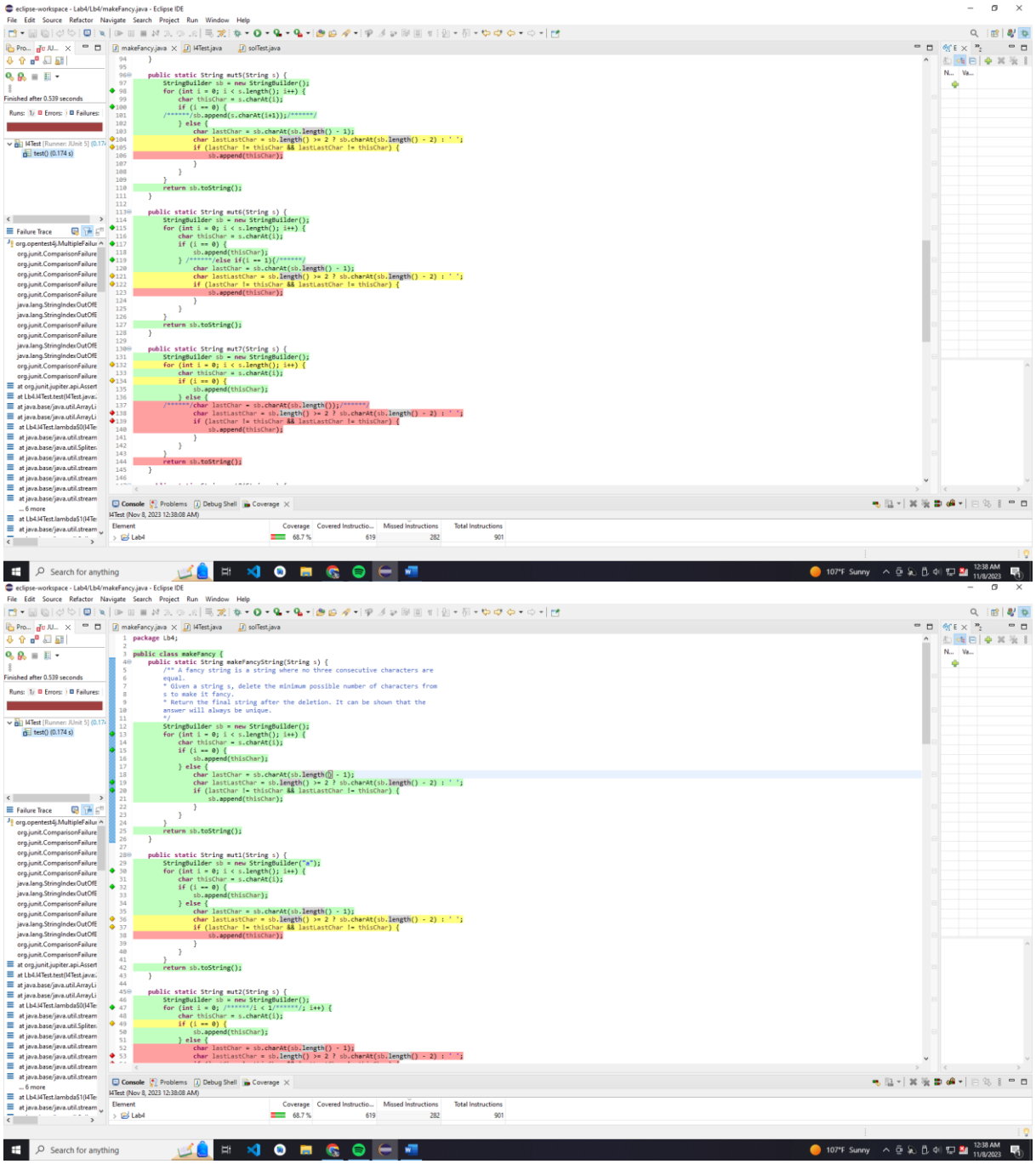
2. Test Results with Traceability

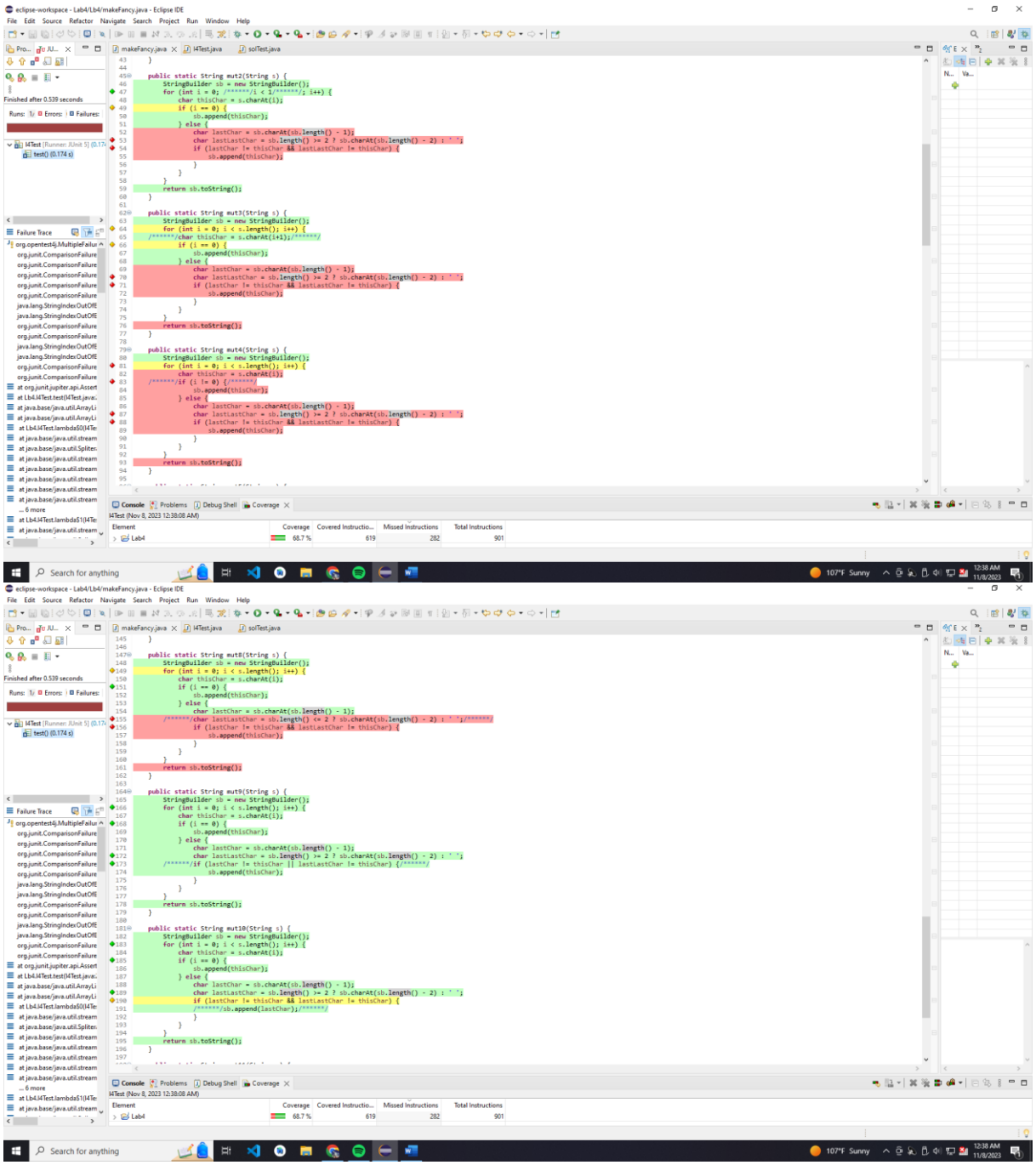
Test ID	Mutation ID	Input	Observed	Observed	Result

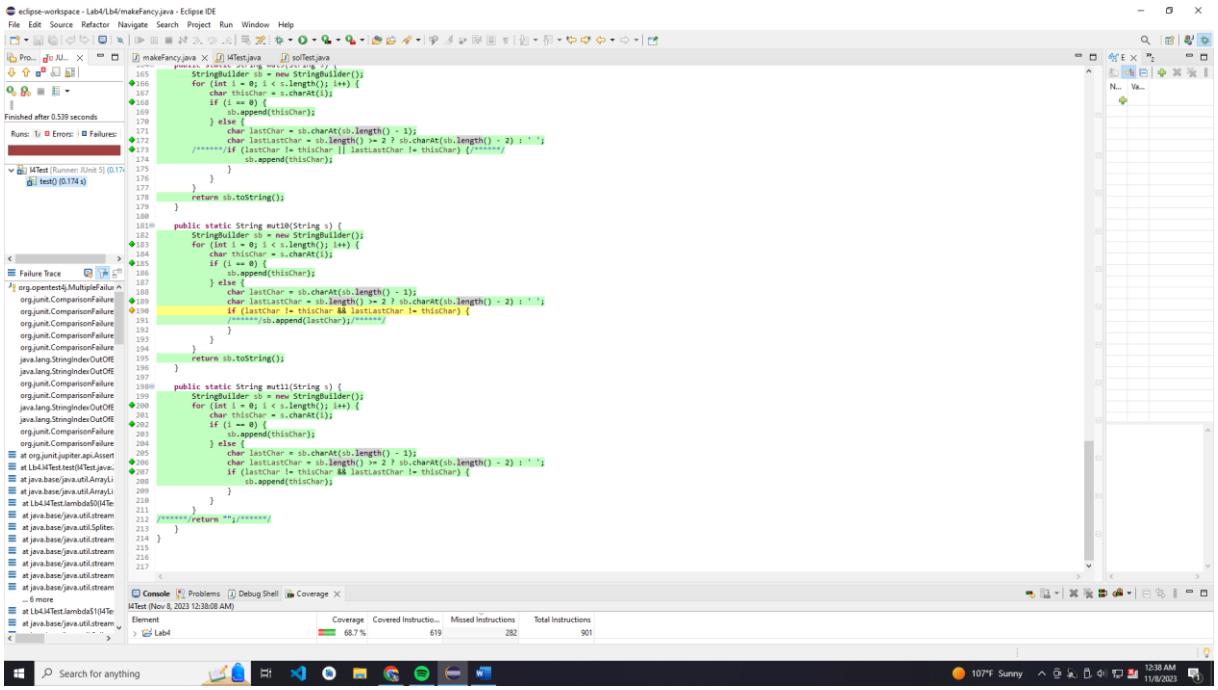
			Output – Original Method	Output – Mutated Method	
1	1	bbbabbbb	ba	ab	Mutant Killed
2	2	aaabaaaa	ab	a	Mutant Killed
3	3	ab	ab	IndexOutOfBounds	Mutant Killed
4	4	aabaaa	ab	IndexOutOfBounds	Mutant Killed
5	5	ab	ab	b	Mutant Killed
6	6	aaabaaaa	ab	a	Mutant Killed
7	7	aaabaaaa	ab	IndexOutOfBounds	Mutant Killed
8	8	aaabaaaa	ab	IndexOutOfBounds	Mutant Killed
9	9	aaabaaaa	ab	aabaa	Mutant Passed
10	10	aaabaaaa	ab	aa	Mutant Killed
11	11	aaabaaaa	ab	“”	Mutant Killed

3. Mutation-Based Coverage Effectiveness

3.1. Compared to Traditional Code Coverage




eclipse-workspace - Lab4/Lb4/makeFancy.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
makeFancy.java
l4Test.java
solTest.java
public static String mut2(String s) {
public static String mut3(String s) {
public static String mut4(String s) {
public static String mut8(String s) {
public static String mut9(String s) {
public static String mut10(String s) {
Console
Problems
Debug Shell
Coverage
l4Test (Nov 8, 2023 12:38:08 AM)
Element
Coverage
Covered Instructio...
Missed Instructions
Total Instructions
Lab4
68.7 %
619
282
901
Search for anything
107°F Sunny
12:38 AM
11/8/2023



I could have easily improved coverage if I included modifications for mutants 3,4,7, and 8 that didn't result in Index Out of Bounds errors. Limiting i to only be less than one also limited the potential coverage in mutation 2.

3.2. Revealing the fault

As labeled in my code as "The golden child mutant", number 9 was the only method to return the correct string. Through my experience with this lab, I have come to understand that the "throw as many darts as you can at the wall and see what sticks" is a somewhat successful testing method. I can foresee, however, that with larger programs, having to make multiple variations of them and having to run them simultaneously may be costly on a machine or time consuming.