**Mutation Testing**

Say, there is a hospital site that lets new users register. It reads the Date of birth or age of the patient. If it is greater than 14, assigns a general physician as their main doctor.  To do so, it invokes the ‘general physician’ function that finds the available doctor.

Now, there might be other functionality. Maybe, patients below 13 get assigned to a pediatrician and so on. But we will only take the age-over-14 case.

This is what the code might look like:

***1)****Read Age*  
***2)****If age>14*  
***3)****Doctor= General Physician()*  
***4)****End if*

Please note that the above lines of code are not specific to any programming language and won’t run. It is just hypothetical.

As a tester, if my data-set is 14, 15, 0, 13 – some random numbers.

The target is to check if the data-set of the 4 values (14, 15, 0, and 3) is adequate to identify all possible problems with this code.

**How does Mutation Testing achieve this?**

First and foremost, you create mutants- variations of the program. A mutant is nothing but a program that is written as a deviation. It contains a self-seeded fault.

Examples are:

* Arithmetic operator replacement
* Logical connector replacement
* Statement removal
* Relational operator replacement
* Absolute value insertion, etc.

These replacements are also called **‘Mutation Operators.’**

**Mutant Examples:**

**Mutant #1: Relational operator replacement**

*1) Read Age*  
*2) If age<14 ‘Changing the > with <’*  
*3) Doctor= General Physician()*  
*4) End if*

**Mutant #2:**

*1) Read Age*  
*2) If age=14 ‘Changing the > with =’*  
*3) Doctor= General Physician()*  
*4) End if*

**Mutant #3:**

*1) Read Age*  
*2) If age>=14 ‘Changing the > with >=’*  
*3) Doctor= General Physician()*  
*4) End if*

**Mutant #4:**

*1) Read Age*  
*2) If age<=14 ‘Changing the > with <=’*  
*3) Doctor= General Physician()*  
*4) End if*

**Mutant #5: Statement Removal**

*1) Read Age*  
*2) If age=14*  
*3) ‘remove the doctor assignment statement’*  
*4) End if*

**Mutant #6: Absolute Value Insertion**

*1) Read Age*  
*2) If age>14*  
*3) Doctor= Mr.X (Absolute value insertion- let’s say X is a pediatrician)*  
*4) End if*

**Mutant #7: Incorrect syntax**

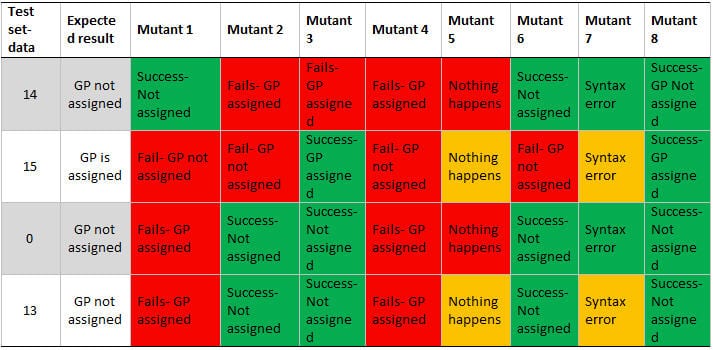
*1) Read Age*  
*2) If age%%14 (incorrect syntax)*  
*3) Doctor=General Physician()*  
*4) End if*

**Mutant #8: Does the same thing as the original test**

*1) Read Age*  
*2) If age> 14 & age>14 ‘means the same thing as age>14’*  
*3) Doctor= General Physician()*  
*4) End if*

Once, all the mutants are created. They are subjected to the test data-set. Our set is 14, 15, 0 and 13. Which of these mutants will our data-set find?

**Find out in the below table:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2016/10/mutation-test.jpg)

As you can see our data value 14 finds failures when it runs against, Mutant 2, 3 and 4. Or, 14 kills mutants 2, 3 & 4. But, it is ineffective against, 1, 6 and 8.

**If your data-set kills all mutants, it is effective.** Otherwise, include more or better test data. It is not necessary for the each value in the data-set to kill all mutants. But together, they should kill all. For example: 14 kills 2, 3 and 4. 15 kills 1, 2 and 4. And, so on.

**What about 5, 7, and 8?**

**Mutant #5 –** is the program instance that will fail irrespective of any data value you give. This is because it will not do any programming for both valid and invalid values.

**Mutant #7** **–** will be a compile error. Or in the case of a scripting language an error that will prevent execution.

**Mutant #8** **–** is the same thing as the main program.

As you can see, the above mutants are not useful at all.

**Therefore, mutants to avoid are:**

* **Syntactically incorrect/‘Still-Born’ mutants. :**You need syntactically correct mutants ONLY. Example: Mutant 7
* **Equivalent Mutants:**The ones that do the exact same thing as the original program. Example: Mutant 8.
* **Trivial Mutant:**Can be killed by any data-set. Example: Mutant 5