Piacente Cristian 866020

Assignment A9 – Mutation Testing

Software Quality, Academic Year 2023-2024, University of Milan – Bicocca

Generate one invalid, one valid-but-not-useful, and one useful mutants for the following program using the provided mutant operators. Assume that a mutant is useful only if it is killed by at most a test case of the provided test suite.

For your valid mutants show the test cases in the provided test-suite that contribute to their killing.

Generate one equivalent mutant, and explain why it is indeed equivalent.

Generate at least mutant that is weakly killed, but not killed, with the provided test-suite.

```
   public class Date {
       private int year;
       private boolean leap;

1      public Date(int y) {
2          if (y <= 1584) {
3              throw new RuntimeException("Invalid year");
           }
4          year = y;
5          leap = (year % 400 == 0);
6          leap = (leap || (year % 4 == 0 && year % 100 != 0));
       }

7      public int lastDayOfMonth(int month) {
8          switch (month) {
9              case 1:
10             case 3:
11             case 5:
12             case 7:
13             case 8:
14             case 10:
15             case 12:
16                 return 31;
17             case 4:
18             case 6:
19             case 9:
20             case 11:
21                 return 30;
22             case 2:
23                 int max = 28;
24                 if (leap) {
25                     ++max;
                   }
26                 return max;
27             default:
28                 throw new RuntimeException("Invalid month");
           }
       }
   }
```

MUTANT OPERATORS:

- A op B --> B op A, where op is a comparison operator
- A op1 B --> A op2 B, where op1 and op2 are 2 different comparison operators
- ++A --> A++
- TRUE --> FALSE
- FALSE --> TRUE
- == --> =
- = --> ==

Piacente Cristian 866020

Assignment A9 – Mutation Testing

Software Quality, Academic Year 2023-2024, University of Milan – Bicocca

TEST SUITE:

```
void test0() {
    Date d = new Date(2024);
    assertEquals(29, d.lastDayOfMonth(2));
}

void test1() {
    Date d = new Date(2023);
    assertEquals(31, d.lastDayOfMonth(3));
}

void test2() {
    Date d = new Date(2024);
    assertEquals(30, d.lastDayOfMonth(4));
}
```

INVALID MUTANT

A mutant is **invalid** if it's not syntactically correct.
We can generate such mutant by applying the mutant operator **= --> ==** on **line 4**.
Here is the resulting **invalid mutant** (only the method containing the mutation is shown for brevity), which would generate a compile error ("error: not a statement"):

```
…
1    public Date(int y) {
2        if (y <= 1584) {
3            throw new RuntimeException("Invalid year");
         }
4        year == y;
5        leap = (year % 400 == 0);
6        leap = (leap || (year % 4 == 0 && year % 100 != 0));
     }
…
```

VALID-BUT-NOT-USEFUL MUTANT

A **valid-but-not-useful** mutant is a syntactically correct mutant which is killed by more than 1 given test case.
We can generate such mutant by applying the mutant operator **y <= 1584 --> y > 1584** on **line 2**.
Here is the resulting **valid-but-not-useful mutant**:

```
…
1    public Date(int y) {
2        if (y > 1584) {
3            throw new RuntimeException("Invalid year");
         }
4        year = y;
5        leap = (year % 400 == 0);
6        leap = (leap || (year % 4 == 0 && year % 100 != 0));
     }
…
```

It's simple to show that it's not a useful mutant: **all the test cases in the test suite throw an exception** ("Invalid year"), so the **1st**, the **2nd and** the **3rd test case** in the test suite **kill the mutant**, which means that the mutant is killed by 3 given test cases.

USEFUL MUTANT

A **useful** mutant is a syntactically correct mutant which is killed by at most 1 given test case.
We can generate such mutant by applying the mutant operator **year % 100 != 0 --> year % 100 == 0** on **line 6**.
Here is the resulting **useful mutant**:

```
…
1    public Date(int y) {
2        if (y <= 1584) {
3            throw new RuntimeException("Invalid year");
         }
4       year = y;
5       leap = (year % 400 == 0);
6       leap = (leap || (year % 4 == 0 && year % 100 == 0));
    }
...
```

It's a useful mutant because with this new definition of leap year **2024 is not a leap year anymore**, so the assertion in the first test case, which checks if the last day of February 2024 is the 29th, will be violated, since the max for February is 28 when leap is false.
It means that only the 1st test case in the test suite kills the mutant, so it's useful.

EQUIVALENT MUTANT
An **equivalent** mutant is a mutant that cannot be distinguished from the original program by any test case.
We can generate such mutant by applying the mutant operator ++max --> max++ on **line 25**.
Here is the resulting **equivalent mutant**:

```
...
7     public int lastDayOfMonth(int month) {
8         switch (month) {
9             case 1:
10            case 3:
11            case 5:
12            case 7:
13            case 8:
14            case 10:
15            case 12:
16                return 31;
17            case 4:
18            case 6:
19            case 9:
20            case 11:
21                return 30;
22            case 2:
23                int max = 28;
24                if (leap) {
25                    max++;
                  }
26                return max;
27            default:
28                throw new RuntimeException("Invalid month");
          }
      }
...
```

It's an equivalent mutant because the only difference between the pre-increment and the post-increment operator is that the last one returns the old value before incrementing, while the first one returns the new value after incrementing; in this case **the value is not used at all in the statement**, so there is no difference in the behavior w.r.t the original program and **there can't exist any test case that kills the mutant.**

WEAKLY KILLED BUT NOT KILLED MUTANT
A mutant is **weakly killed but not killed** if it produces a different intermediate state w.r.t the original program, but not a difference in the final result.
We can generate such mutant by applying the mutant operator year % 400 == 0 --> year % 400 != 0 on **line 5**.
Here is the resulting **weakly killed but not killed mutant**:

```
...
1     public Date(int y) {
2         if (y <= 1584) {
3             throw new RuntimeException("Invalid year");
          }
4         year = y;
5         leap = (year % 400 != 0);
6         leap = (leap || (year % 4 == 0 && year % 100 != 0));
      }
```

…
The mutant is weakly killed but not killed, with the provided test-suite, because **the intermediate state will differ from the original program state after executing line 5**, since the equal operator has been replaced with its negation, and the mutant is not killed because February (the only month with a number of days that depends on whether the year is leap or not) is only used in the first test case, where the expression at line 6 is still evaluated as true: in the original program the OR expression at line 6 is true because 2024 % 4 == 0 && 2024 % 100 != 0 is true, while in the mutant it's true because 2024 % 400 != 0 is true.
It means that **the result is the same for all given test cases**, so the mutant is not killed.