

# Dynamic Analysis: Testing

John Businge

[john.businge@unlv.edu](mailto:john.businge@unlv.edu)

# Introduction

- Dynamic Analysis verifies properties of a system during execution/runtime.
- Testing Analysis is one example of Dynamic Analysis
  - Unit tests, integration tests, system tests, and acceptance tests use dynamic testing

# Testing

- Tests are your life insurance! (OORP, p. 149)
- Tests are essential to assuring the quality of refactoring/code change activities.
- Write Tests to Enable Evolution (OORP, p.153)
  - Good tests can find bugs in your artifact
  - Tests can also detect unwanted behaviour
- You can also write tests to understand a part of a system (OORP, p.179)

# Unit Testing

- In this session, we focus on Unit Testing.
- There are other types of testing (Integration, Performance, Security, etc.)
- It does not mean that Unit Testing is more important, but those are the tests we can more easily automatize and benefit from tool support.

# Quality of a Test Suite

- How do you know if your unit test cases are good enough?
- Are they really testing the application?
- When do we stop testing?

**Solution:** Test Coverage!

# Test Coverage

$$\text{Coverage} = \frac{\text{Number of Covered Items}}{\text{Total number of items}} \times 100\%$$

- Statement (Line, or Code) Coverage
- Branch (Condition) Coverage
- Path Coverage
- Mutation Coverage

# Example: a function to test

```
int foo(int input, bool b1, bool b2, bool b3){  
    int x = input;  
    int y = 0;  
    if(b1)  
        x++;  
    if(b2)  
        x--;  
    if(b3)  
        y=x;  
    return y;  
}
```

# Statement/Line/Code Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3){  
    int x = input;  
    int y = 0;  
    if(b1)  
        x++;  
    if(b2)  
        x--;  
    if(b3)  
        y=x;  
    return y;  
}
```



# Statement/Line/Code Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3) {  
    int x = input;  
    int y = 0;  
    if(b1)  
        x++;  
    if(b2)  
        x--;  
    if(b3)  
        y=x;  
    return y;  
}
```

100% Statement Coverage

# Statement/Line/Code Coverage

Test Case(s)

```
ASSERT foo(0, faulse, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3) {  
    int x = input;  
    int y = 0;  
    if(b1)  
        x++;  
    if(b2)  
        x--;  
    if(b3)  
        y=x;  
    return y;  
}
```

$$\frac{8}{9} \times 100\% = 88.9\%$$

~~100% Statement Coverage~~

# Branch/Condition Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
int foo(int input, bool b1, bool b2, bool b3) {  
    int x = input;  
    int y = 0;  
    if(b1)  
        x++;  
    if(b2)  
        x--;  
    if(b3)  
        y=x;  
    return y;  
}
```

**50% Branch Coverage**

# Branch/Condition Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;  
Assert foo(0, false, false, false) == 0;
```

New Test

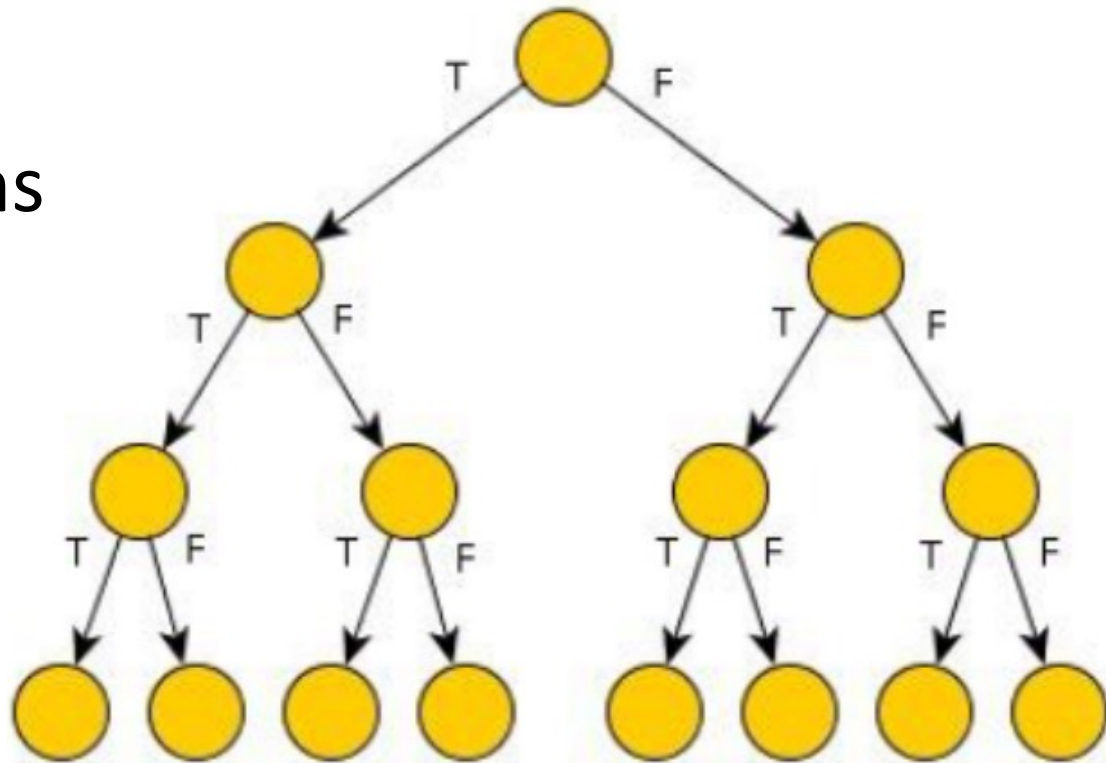
```
int foo(int input, bool b1, bool b2, bool b3){  
    int x = input;  
    int y = 0;  
    if(b1)  
        x++;  
    if(b2)  
        x--;  
    if(b3)  
        y=x;  
    return y;  
}
```

100% Branch Coverage

# Path Coverage

Paths for three “if” each can be either true (T) or false (F)

# 8-Paths

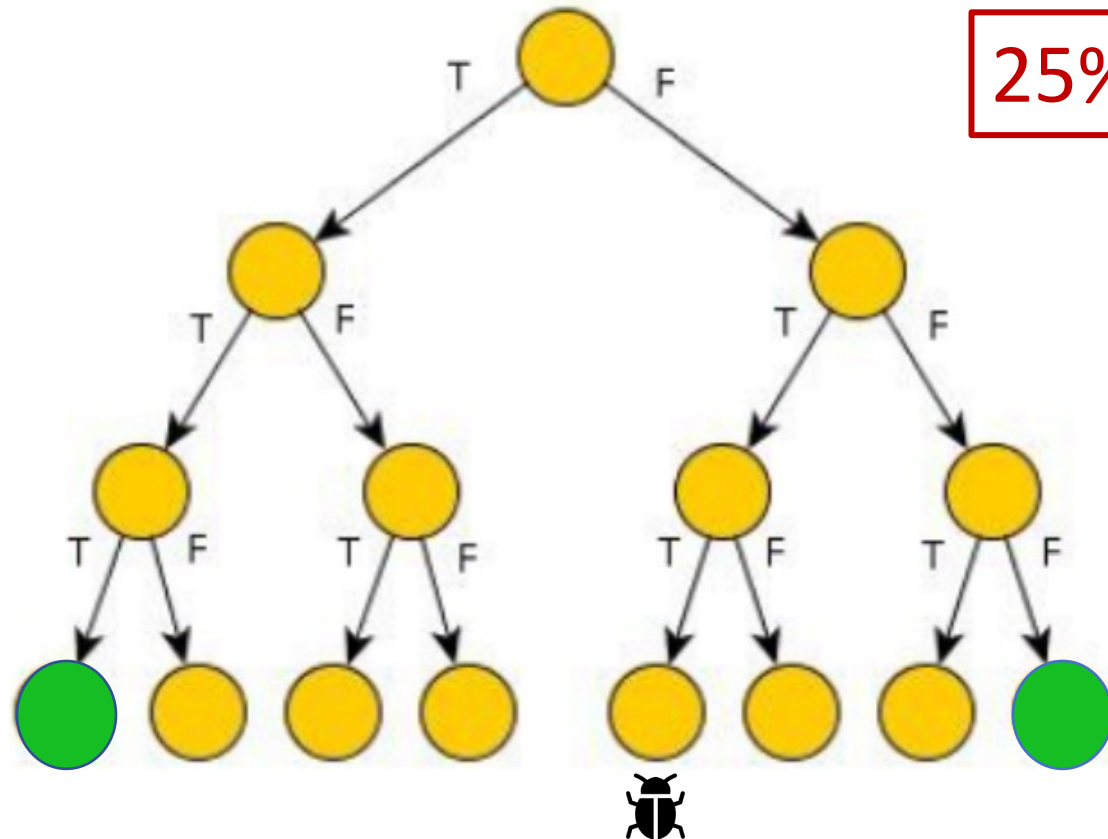


# Path Coverage

Test Case(s)

```
ASSERT foo(0, true, true, true) == 0;
```

```
ASSERT foo(0, false, false, false) == 0;
```

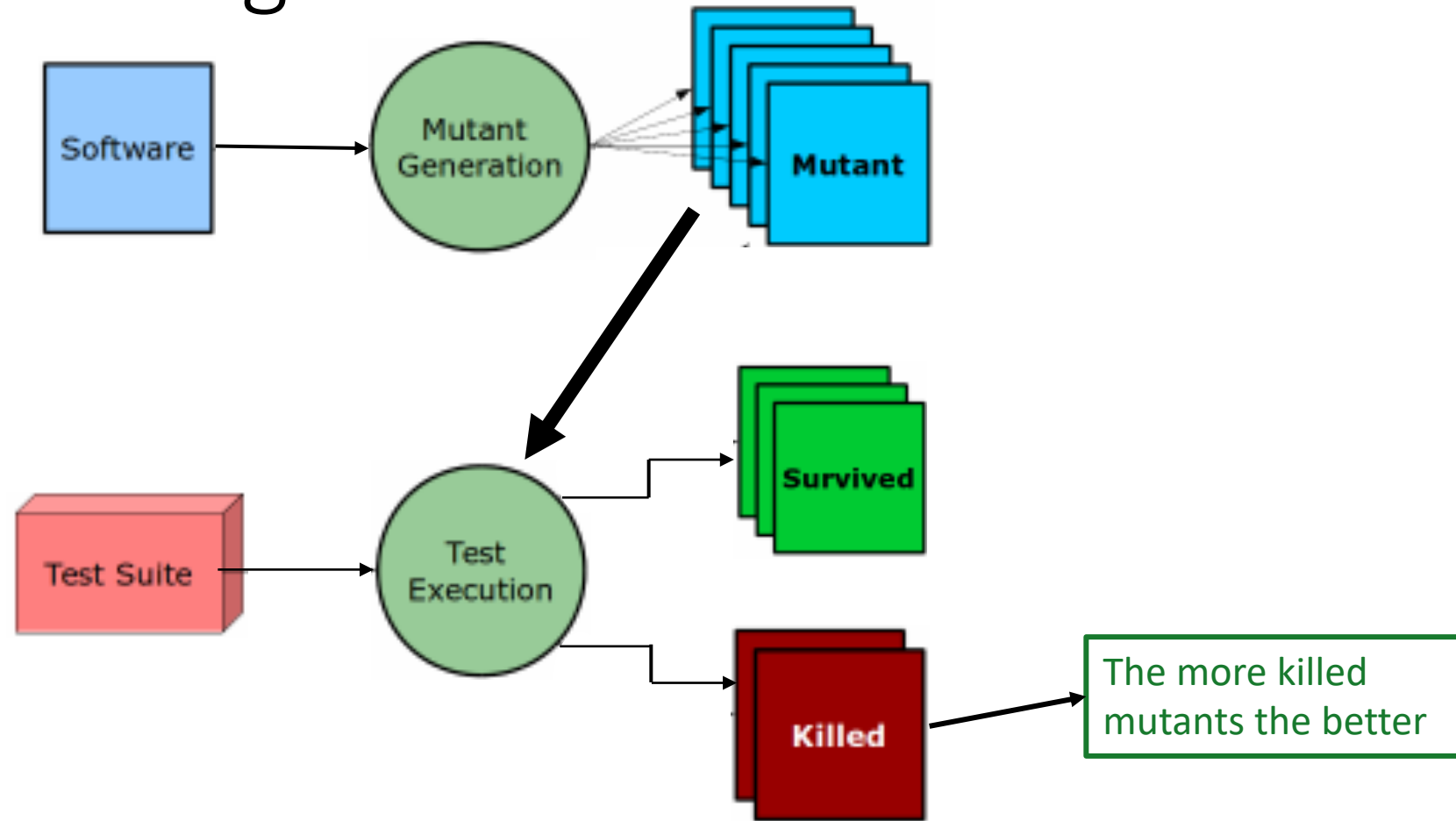


# Mutation Testing

- The steps for Mutation Testing are basically:
  - Make small changes to the code (Mutants).
  - Each change is a different mutant.
  - For each mutant, run the test cases:
    - If one test fails, it means your test was good enough to detect the changes (the mutant is killed).
    - If all tests pass, it means your tests did not detect the changed behaviour (the mutant survives).
- Therefore, the more mutants you kill, the better

$$\text{Mutation Coverage} = \frac{\text{Number of Killed Mutants}}{\text{Total number of Mutants}} \times 100\%$$

# Mutation Testing



$$\text{Mutation Coverage} = \frac{\text{Number of Killed Mutants}}{\text{Total number of Mutants}} \times 100\%$$



# Mutation Testing: Small Example



Original

```
int f(bool a, bool b){  
    if(a && b) return 1;  
    else return 0;  
}
```

Test Case

```
void testf(){  
    assert f(true, true)==1;  
    assert f(false, false)==0;  
}
```

Mutant

```
int f(bool a, bool b){  
    if(a || b) return 1;  
    else return 0;  
}
```

Mutant Survives  
the Test Case

# Mutation Testing: Small Example



Original

```
int f(bool a, bool b){  
    if(a && b) return 1;  
    else return 0;  
}
```

Test Case

```
void testf(){  
    assert f(true, true)==1;  
    assert f(false, false)==0;  
}
```

Mutant

```
int f(bool a, bool b){  
    if(a || b) return 1;  
    else return 0;  
}
```

**Missing Assertions that  
Could Kill this Mutant**

```
assert f(false, true)==0;  
assert f(true, false)==0;
```

# Mutation Coverage

- Assess how good your test cases are at catching faults by introducing defects into the source code.
- More reliable metric to validate test suite effectiveness.

# Testing Coverage for the Project

- It is required to show coverage for your Project (in both the Intermediate and the Final Report)
  - At least Statement Coverage, but Branch Coverage is better.
  - You should show the chosen coverage before the refactoring/change and after (where hopefully you also added new tests).
- There is no set coverage limit to reach for the project.
- But if your project has very low coverage, you better have a good explanation for that.
- Focus on increasing the coverage for the system parts that will be affected by your refactoring/change.