COMS W4156 Advanced Software Engineering (ASE)

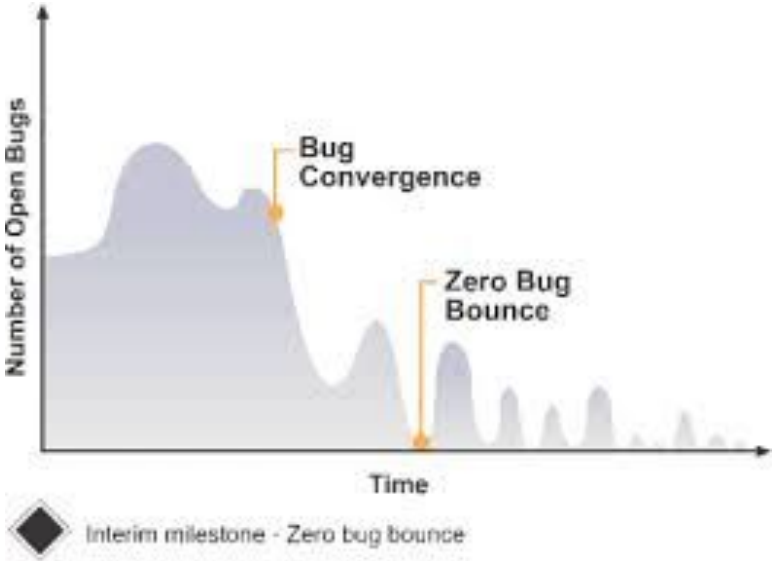
November 4, 2021

shared google doc for discussion during class

Refresher: How Do We Know When We Are Done Testing?

- You're *never* done if you keep changing the code (regression testing)
- This is why many software organizations "freeze" a planned release build a week or more in advance, with no changes allowed except bug fixes (with each set of repairs followed by full re-testing)

Zero bug bounce: Rate of finding new bugs is stable and low (approaching zero), reached point of diminishing returns



But How Do We Know We Have Done Enough Testing To Release?

Functional testing:

1. Your organization always releases according to schedule no matter what
2. Zero (or close to zero) bug bounce
3. Coverage! You are never truly done testing, but some stopping points are better than others

Warning: blackbox testing sometimes called “functional testing” to contrast with whitebox “structural testing”, but whitebox is still considered “functional testing” in contrast to NON-functional testing

NON-Functional testing:

Security!

Performance, Scalability

Interoperability, Portability

UIUX, Accessibility, Localization (when applicable)

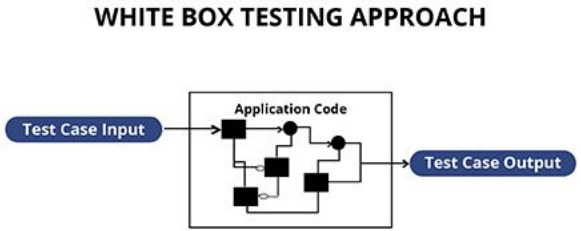
others...

(discussed later in semester)

What is Coverage Covering?

Blackbox testing focuses on inputs and outputs, without concern for how the software under test transforms the inputs into outputs. Equivalence partitioning intends to “cover” the input space (and sometimes output space), but that’s not what is usually meant by coverage

Coverage generally refers to some program representation graph or “structure” (whitebox) and exercising every node and/or transition in the graph



Intuition behind coverage: What if some of the code has never been executed? How can you know whether it works?

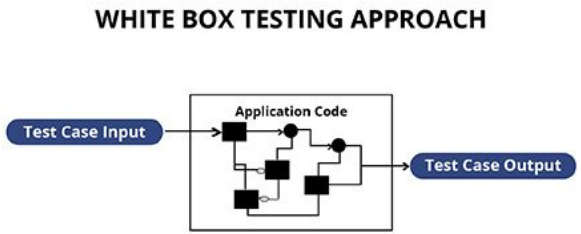
(Why is the code there if it is not needed for any of the user stories or use cases? Sanity check: comment out all code not covered by blackbox testing, does it matter?)

What is Coverage Covering?

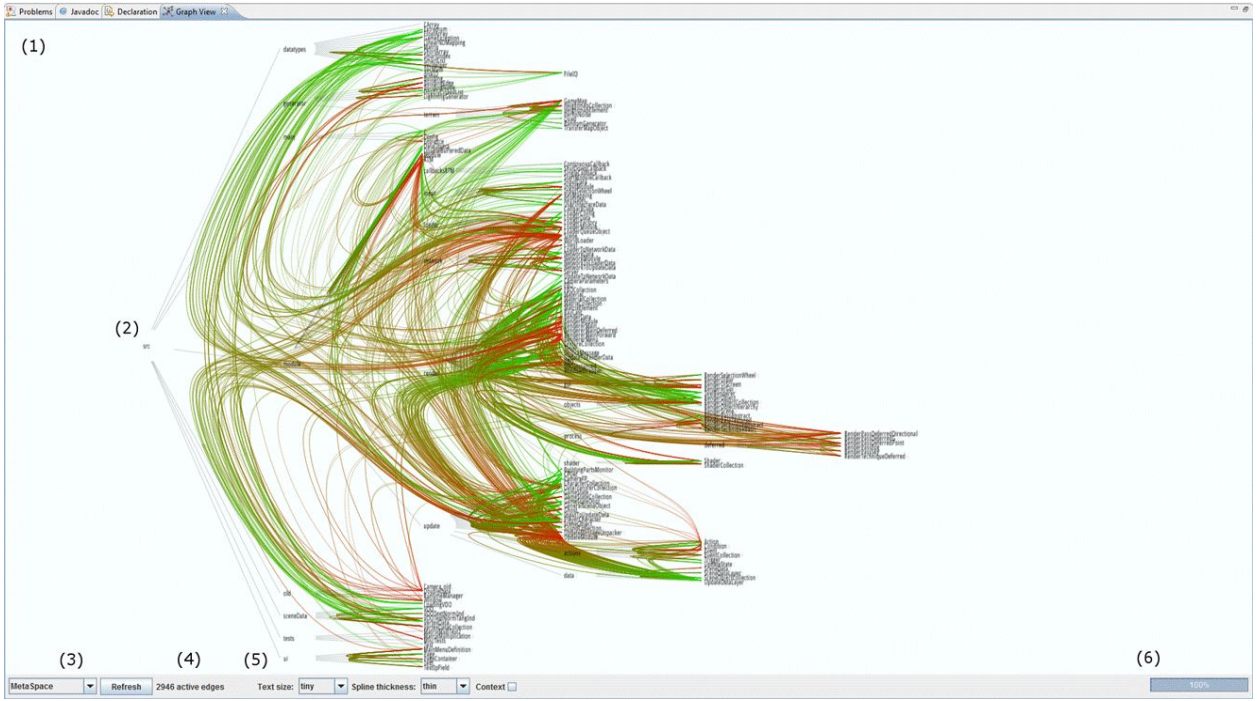
The conventional graph to cover is the set of control-flow execution paths = the sequence of instructions executed for a given input to produce the corresponding actual output

Ideally exercise (“cover”) *every* execution path through the program, but there are usually far too many paths

Other graphs: call graph, program dependence graph,
data flow graph, state transitions, ...



Interprocedural Control Flow Paths



An execution path from beginning to end of the entire program runs through all the methods invoked during that execution of the program

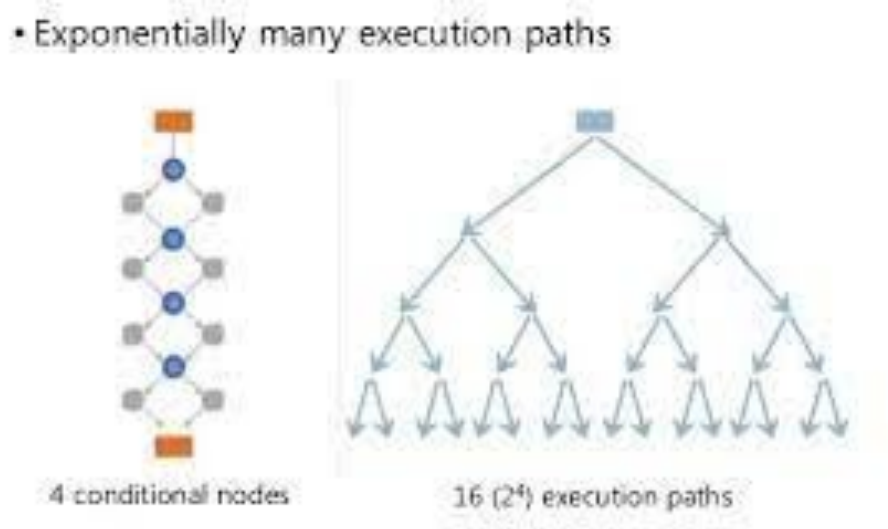
An interprocedural call graph represents methods as nodes and method calls as edges, it's smaller than a control flow graph through those calls - but still need *many* executions to, eventually, call every method

Intraprocedural Control Flow Paths

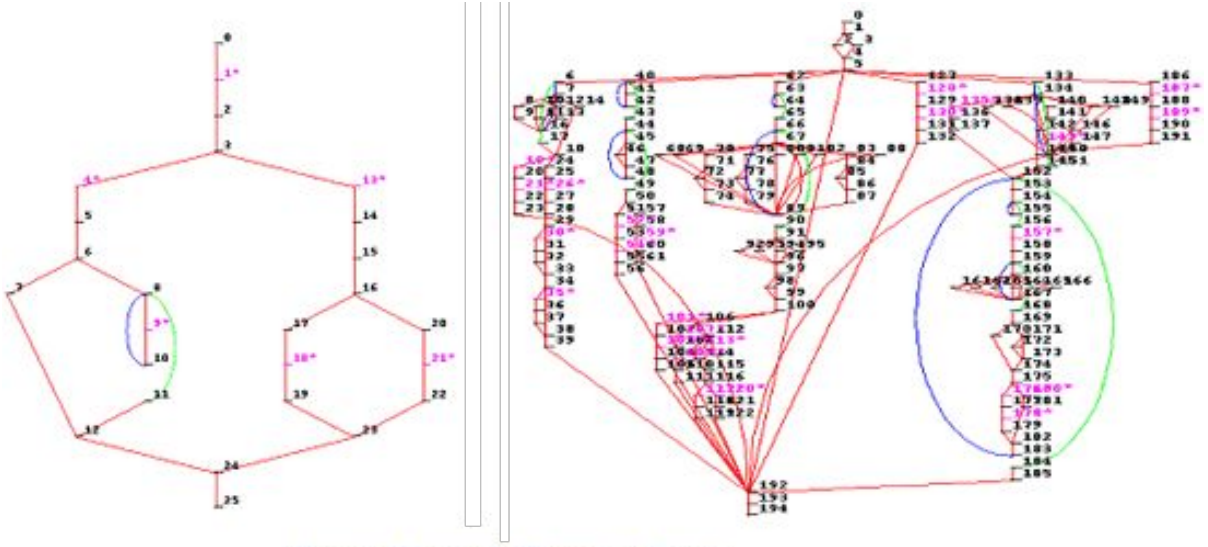
An execution path through an individual method includes all statements invoked during that execution

Every blackbox unit test of a method executes *some* execution path through that method

But how many blackbox unit tests would we need to execute *every* execution path through the method?

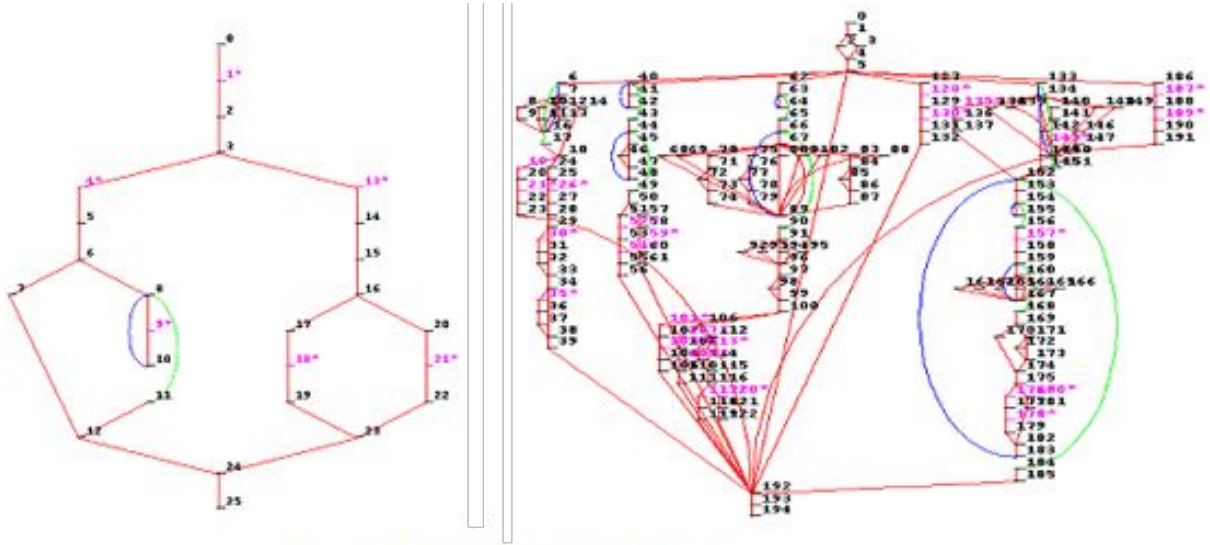


More Intraprocedural Control Flow Paths



Simple Code vs. Complex Code

Cyclomatic Complexity



Simple Code vs. Complex Code

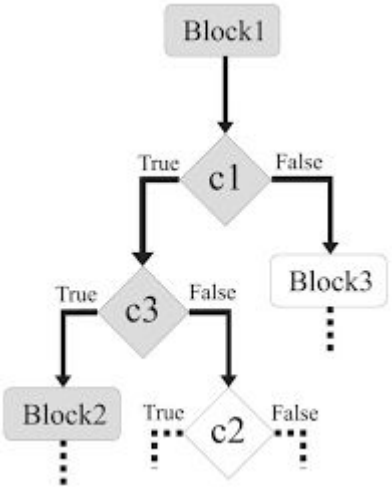
Cyclomatic complexity = number of conditions + 1

Minimum number of test cases required to achieve branch coverage

Branch Coverage

Since it is not feasible to test *all* execution paths for non-trivial programs, or even for non-trivial individual functions, the tester needs to select a practically small number of paths that, collectively, are most likely to reveal bugs

So most coverage tools support if-else *branch coverage*, which checks that the test suite executes both true and false for every branch, rather than path coverage (much simpler graph)



Statement Coverage

Some tools support only statement coverage or function coverage instead of branch coverage - lesser criteria that are easier to achieve but miss more code

Branch coverage normally implies statement coverage - except when there is dead (unreachable) code, but statement coverage does not imply branch coverage

Branch Coverage

At least one test where the branch is true - the if part is executed, and

At least one test where the branch is false - the if part is not executed and the else part, if any, is executed

```
if (condition)
  { do something }
else
  { do something else }
// some other code

if (condition)
  { do something }
// some other code
```

Branch Coverage

This code has cyclomatic complexity 14

14 sounds high, but this code is so simple: Do we really need to test all the branches? To achieve 100% branch coverage → yes

```
String getMonthName (int month) {
    switch (month) {
        case 0: return "January";
        case 1: return "February";
        case 2: return "March";
        case 3: return "April";
        case 4: return "May";
        case 5: return "June";
        case 6: return "July";
        case 7: return "August";
        case 8: return "September";
        case 9: return "October";
        case 10: return "November";
        case 11: return "December";
        default: throw new
IllegalArgumentException();
    }
}
```

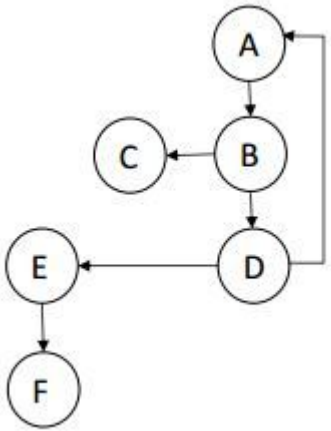
Branch Coverage can be Complicated to Achieve

Both of these examples have cyclomatic complexity 5

```
String getWeight(int i) {
        if (i <= 0) {
            return "no weight";
        }
        if (i < 10) {
            return "light";
        }
        if (i < 20) {
            return "medium";
        }
        if (i < 30) {
            return "heavy";
        }
        return "very heavy";
}
```

```
int sumOfNonPrimes (int limit) {
    int sum = 0;
    OUTER: for (int i = 0; i < limit; ++i) {
                if (i <= 2) {
                        continue;
                }
                for (int j = 2; j < i; ++j)  {
                        if (i % j == 0) {
                                continue OUTER;
                        }
                }
                sum += i;
        }
        return sum;
}
```

Loop Coverage



Branch coverage is too limited for loops → traverse loop or not

Loop statements can be considered multi-way branches, perhaps ∞ many, but we need to stop somewhere so a typical rule of thumb is 0, 1, 2, t, m iterations to consider a potentially infinite loop to be covered. t = typical, m = many

If there is a specified maximum number of iterations (for loop instead of while loop), then 0, 1, 2, max, max+1

Loop Coverage

Rationale for 0: Is some action taken in the body that should also be taken when the body is not executed?

Rationale for 1: Check lower bound on number of times body may be executed.

Rationale for 2: Check loop re-initialization

Rationale for t: This is what will typically occur…

Rationale for m: Atypical but valid...

Rationale for max: Check upper bound on number of times body may be executed.

Rationale for max+1: If maximum can be exceeded, what behavior results?

What Do Branch Coverage Tools Actually Do With Loops?

```
while True:
    if cond:
        break
    do_something()
```

This while loop will never exit normally, so it cannot take both of its “possible” branches

For some such constructs, such as “while True:” and “if 0:”, some coverage tools (e.g., coverage.py) understand what is going on - this will not be marked as a partial branch

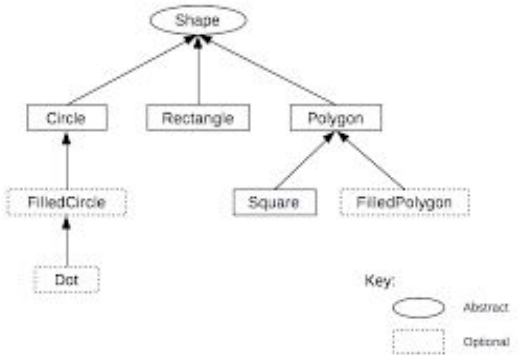
```
i = 0
while i < 999999999:  # pragma: no branch
    if eventually():
        break
```

This while loop will never complete because the break will always be taken at some point

Coverage tools cannot work this out on their own, but often support annotations to avoid flagging branches known to be partial

Limitations of Branch Coverage

```
function F(n)  if n = 0
   return 0  if n = 1
   return 1  else
   return F(n-1) + F(n-2)
```



Most coverage tools do not support branches more sophisticated than if-else very well, if at all

Loop ~= recursion

Polymorphism in object-oriented languages allows “hidden” branches, one for each class of object that can be substituted

“Hidden” branches may also be created via in-lining and other compiler optimizations

Condition Coverage

Multiple Condition Coverage (MCC) vs. Modified Condition/Decision Coverage (MC/DC)

Rarely used outside safety-critical software

MCDC: each individual condition is true at least once and false at least once [linear]

MCC: conditions must evaluate to T or F in all combinations! [exponential]

Short circuit: Only need to test conditions that independently affect decision's outcome

```
(false AND anything-else)
```

```
if (A and B)
  { do something }
else
  { do something else }
```

```
if (A or B)
  { do something }
else
  { do something else }
```

Example

```
boolean purchaseAlcohol
(int buyerAge, int ageFriend)
{
    boolean allow = False;
    if ((buyerAge>=21) or
        (ageFriend>=21))
    { allow = True; }
    return allow;
}
```

Assert purchaseAlcohol(25,25) == True
gives 100% *statement* coverage

Assert purchaseAlcohol(25,25) == True
Assert purchaseAlcohol(20,20) == False
gives 100% *branch* coverage

But *both* the buyer *and* the friend must be 21 or older

Does MCDC coverage detect the bug?

```
boolean purchaseAlcohol
(int buyerAge, int ageFriend)
{
    boolean allow = False;
    if ((buyerAge>=21) or
        (ageFriend>=21))
    { allow = True; }
    return allow;
}
```

purchaseAlcohol(25,25) == True covers T, T

purchaseAlcohol(20,20) == False covers F, F

purchaseAlcohol(25,20) == False covers T, F

purchaseAlcohol(20,25) == False covers F, T

Does a set of test cases fulfilling MCC coverage (truth table) detect the bug?

P	Q	$\neg P$	$P \rightarrow Q$	$\neg P \wedge (P \rightarrow Q)$
T	T	F	T	F
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

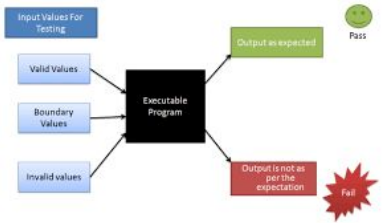
Achieving High Branch Coverage

The tester should devise a set of tests that achieve high branch coverage, ideally 100% but more realistically goal set as 80% or 90%. This measures “test suite adequacy”

Many branches are covered “for free” by the equivalence classes tested during blackbox testing. Extending equivalence classes with *boundary analysis* adds more “for free” branches covered

For example, one test case *inside* an equivalence class and another test case *outside* that equivalence class will typically exercise the true and false branches, respectively, of at least one condition somewhere in the code

What would it mean if there was no such condition?



Forcing Coverage

In general, if we track coverage during execution of blackbox testing, then we can find out what has **not** been covered yet

To add a test that *forces* a not-yet-covered branch, we may need to replace even standard libraries with test doubles

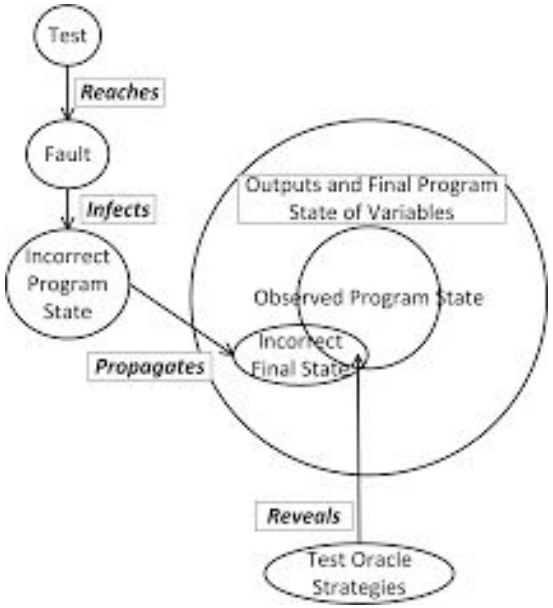
```
void foo () {
    lots of code

    x = outside-call(y);

    if ( x > y ) { do this }
    else { do that }

    lots more code
}
```

➢ How would we cover every statement and/or every branch in a given method if we only used full-system testing, no unit testing?

Consider How System Testing Finds Bugs

First, at least one system-level test must *reach* the location (or locations) in the code containing the bug. Or reach the location(s) where the code ought to be, in cases where the bug is due to missing code - this could be anywhere in the program!

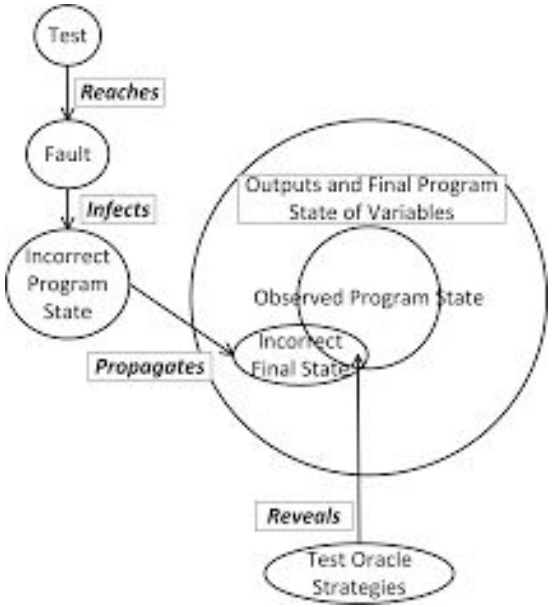
After that part of the code has been executed (or the entire program executed when code is missing), the state of the program must be incorrect with respect to achieving its responsibilities to the program design. The program state has been *infected*



How System Testing Finds Bugs

This infected program state must *propagate* through later code, if any, executed by the same test case, to cause some externally visible state or output to be incorrect

Finally, the test case assertions (or human-checked logs, print statements, etc.) must *reveal* the incorrect aspects of the program state or output. Even if the buggy state is “visible”, it won’t actually be seen unless you look for it

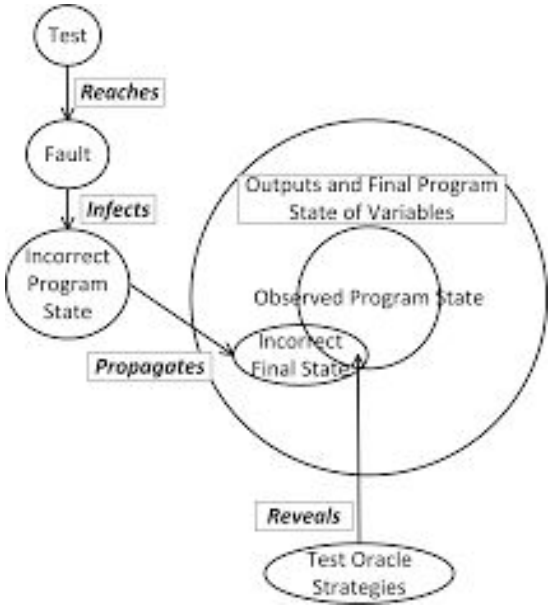


How System Testing Finds Bugs

This is why structural (whitebox) test coverage is so important:

If your test cases never reached part of the code, then your test cases never reached the bug in that part of the code (or could determine that none of your test cases reached the missing code, because it's missing)

But your users will!



How Is Branch Coverage Recorded?

```
foo (int Y) {
    if (Y<=0)
        { Y = -Y;
          log(foo, 'if', true); }
    else // added
        { log(foo, 'if', false);  }
    logcount(foo, 'while', 0);
    while (Y>0) {
        logcountadd(foo, 'while', 1);
        do something;
      Y = Y-1;
    }
}
```

Manually adding logging code to record coverage as the program executes would be very tedious for non-trivial programs, and might confuse debugging

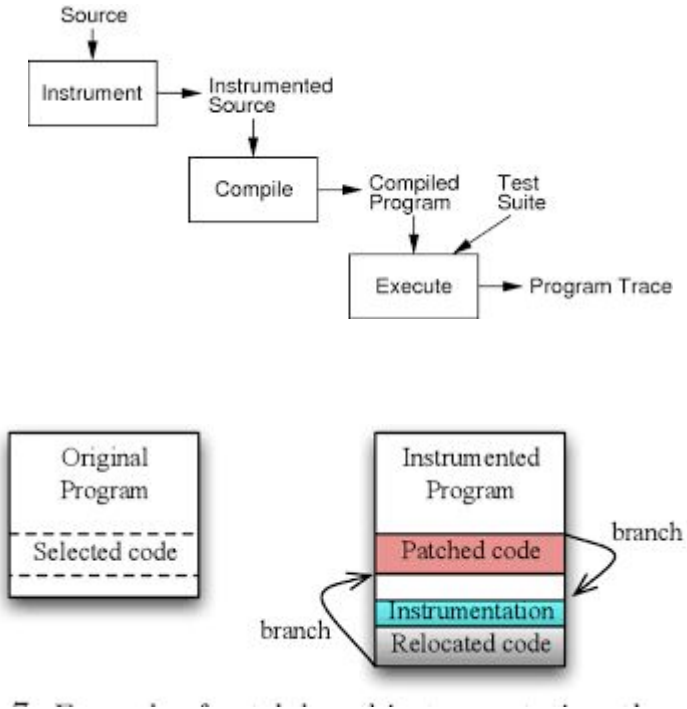
Has to be removed after testing or will add overhead to production

Coverage Tools Instrument Code

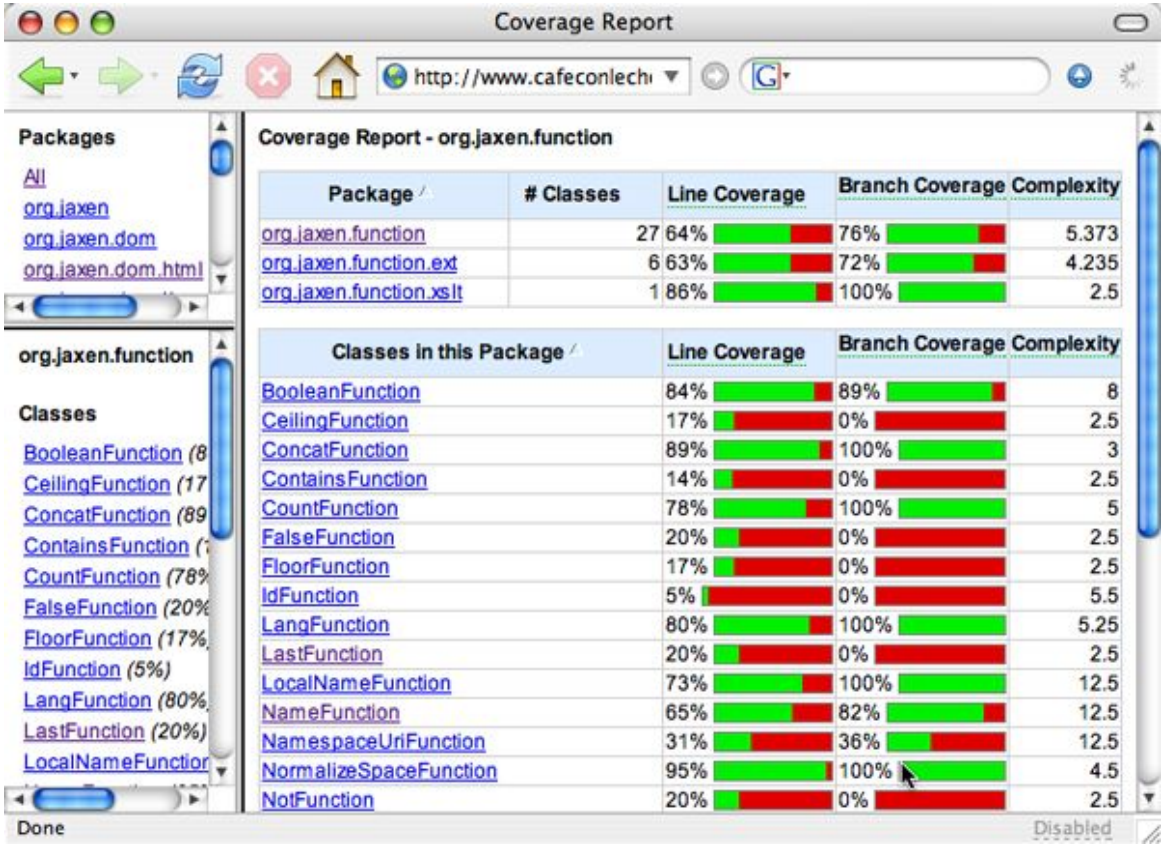
Better to use coverage tools (or compiler options) that automatically insert coverage-checking code at the source, bytecode or binary level

Coverage tools *instrument* the code to track what has already been covered versus not yet covered as the test suite executes

If entire program is instrumented (only during testing!), can accumulate coverage during regular blackbox unit, integration and system testing as well as during tests specifically designed to achieve coverage



Coverage Report



At end of test suite execution, coverage tools inform the tester what percent has been covered (0 <= covered <= 100%) and may annotate (e.g., in an IDE) which specific branches are not yet covered

So Why Not Use Only Whitebox Testing?

Whitebox testing cannot catch errors of *omission*, where code to implement required functionality does not exist

It is not hard to create a trivial program that does not fulfill any of our requirements, but our tests cover all the branches!

```
main ()
{
    return;
}
```

Team Project

Assignment T3: First Iteration due November 15

Assignment T4: First Iteration Demo due November 19 (its ok to do the demo before November 15)