Lecture Notes
November 12, 2020

Working as a team: When are you done testing?

You're never truly done, but some stopping points are better than others.

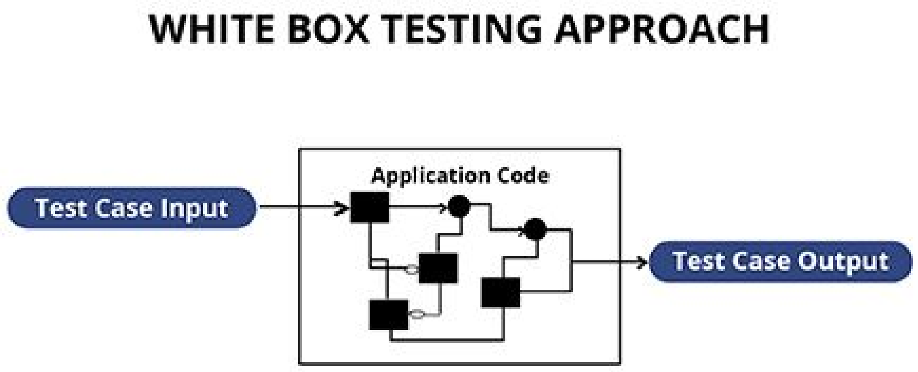
Functional (blackbox) testing focuses on inputs and outputs, without concern for how the software under test (SUT) transforms the inputs into outputs.

We might decide we are done when the code has passed acceptance tests corresponding to all the conditions of satisfaction for all use cases. Ready to ship! ("Good developers develop, great developers ship.")

But… but… but…

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 use cases?)

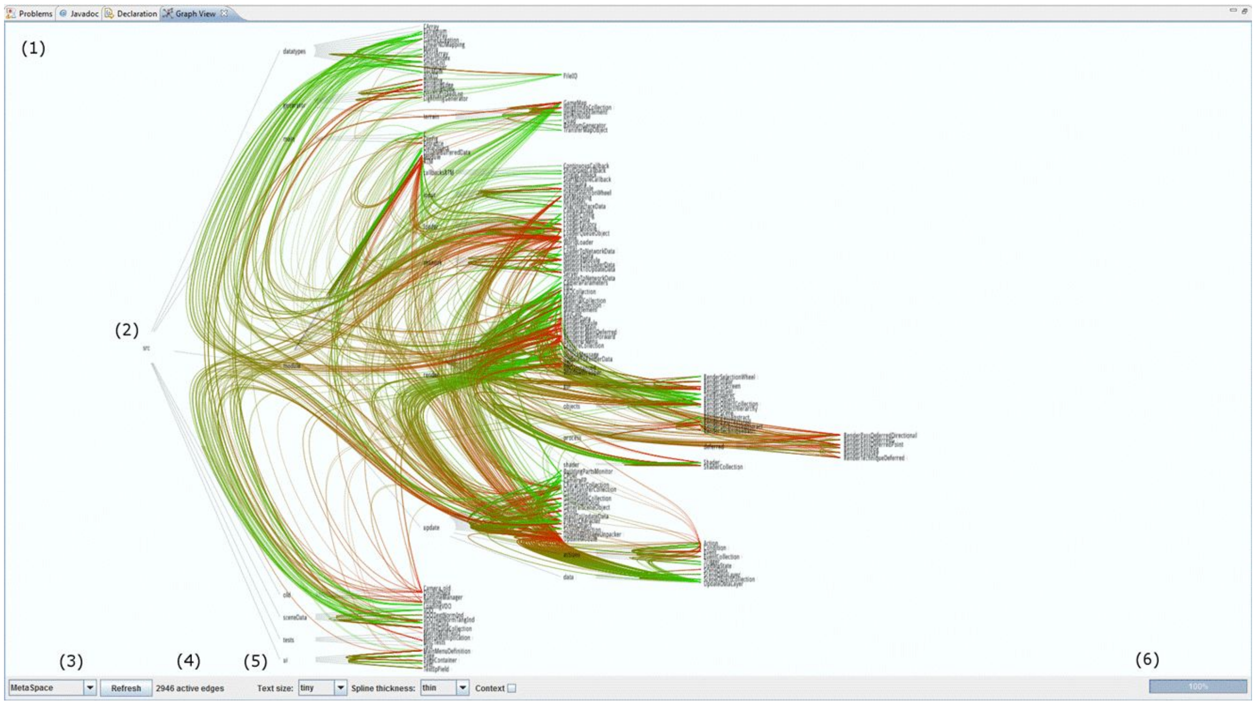
Structural (whitebox) testing focuses on “*coverage*”, ideally exercising all the execution paths (sequences of instructions) that transform inputs into outputs.



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

Functional vs. structural tests concern the different techniques used for choosing inputs, these are not different levels of tests like unit, integration, system.

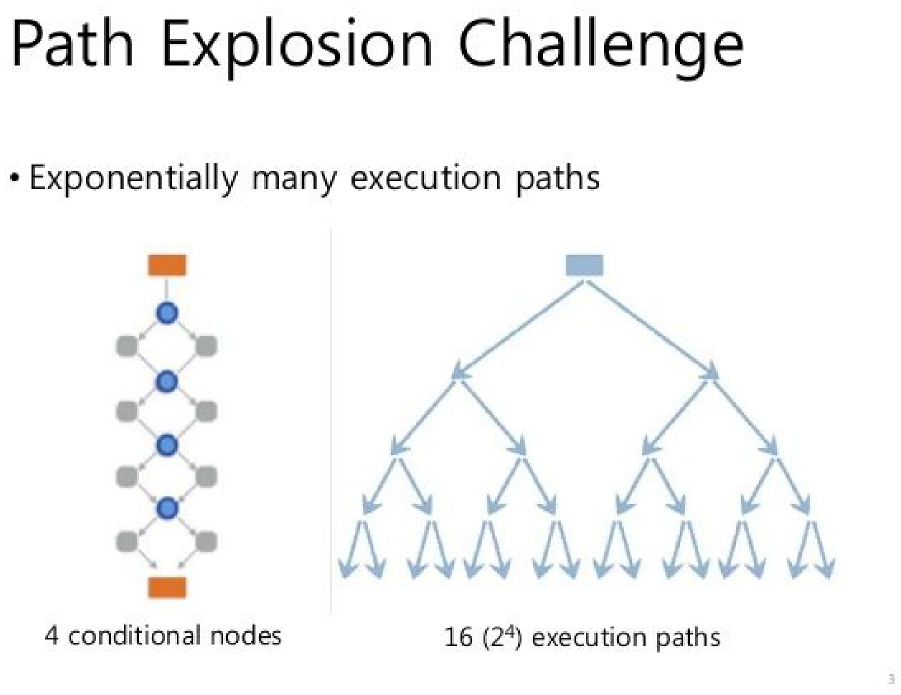
An execution path from beginning to end runs through all the methods invoked during that execution. An interprocedural control flow graph (CFG) is a call graph that represents methods as nodes and invocations as edges. Call graphs can be huge!



MetaSpace
2946 active edges
Text size: tiny
Spline thickness: thin
Context
100%

So logging of execution paths is normally restricted to intraprocedural, rather than interprocedural. This only considers paths through instructions from the beginning of a method to its end, not the combinatorics between methods. Control flow graphs within a method represent basic blocks as nodes and conditional branches as edges.

But even the number of execution paths within a single method can be too many if there are many branches (cyclomatic complexity = number of branches + 1).



Path Explosion Challenge

• Exponentially many execution paths

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.

This means there is at least one test where the branch is true (the if part is executed) and at least one test where it 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
```

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

Branch coverage normally implies statement coverage - except when there is dead (unreachable) code.

Loop coverage is a special case of branch coverage.

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

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

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.

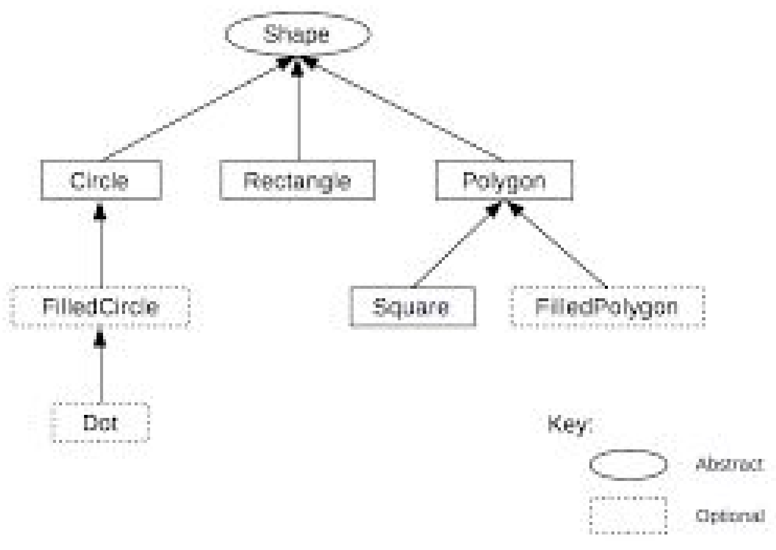
Rationale for 2: Check loop re-initialization

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

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

Most coverage tools do not support more sophisticated branches very well:

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 is not much harder to implement than branch coverage, but rarely used outside safety-critical software. Each individual condition is true at least once and false at least once.

Why would we want condition coverage?

```
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 branch coverage detect the bug?

For condition coverage, conditions must evaluate to T or F in all combinations (truth table)

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

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

A	T	T	F	F
B	T	F	T	F

```
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 condition coverage detect the bug?

The tester should devise a set of tests that achieve high coverage, e.g., 90%, for whatever coverage metric is chosen

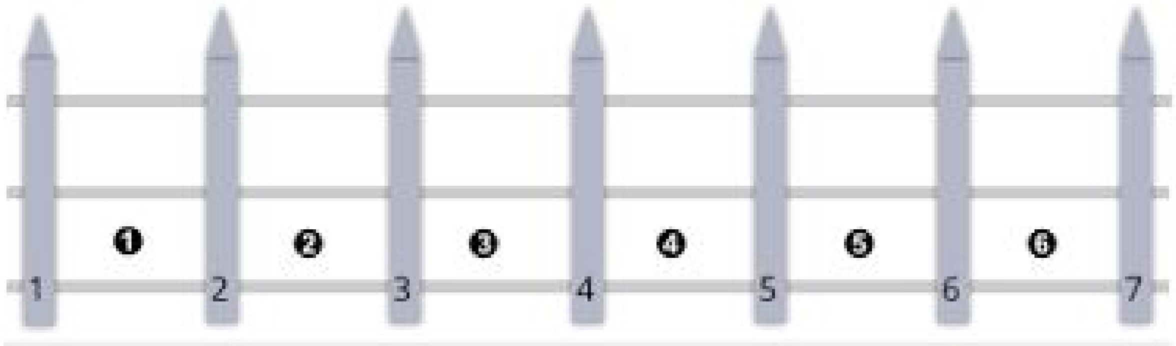
Many branches are covered “for free” by the equivalence classes tested during black box testing

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?

Extending equivalence classes with *boundary analysis* adds more “for free” branches covered

Errors often occur at the *boundaries* of ranges or ordered sets that correspond to equivalence classes: off-by-one error, fencepost error

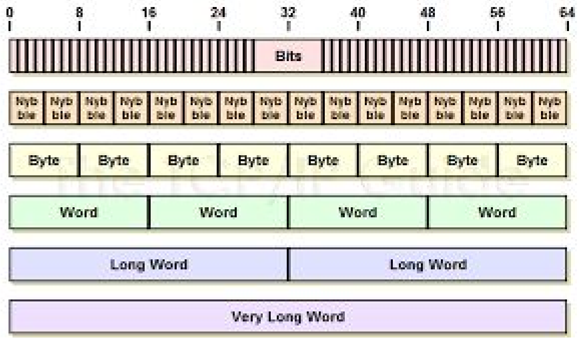


Seven fenceposts are required to complete a six-section fence

Boundary analysis aims to check the boundaries of each equivalence class in addition to checking a value “in the middle”

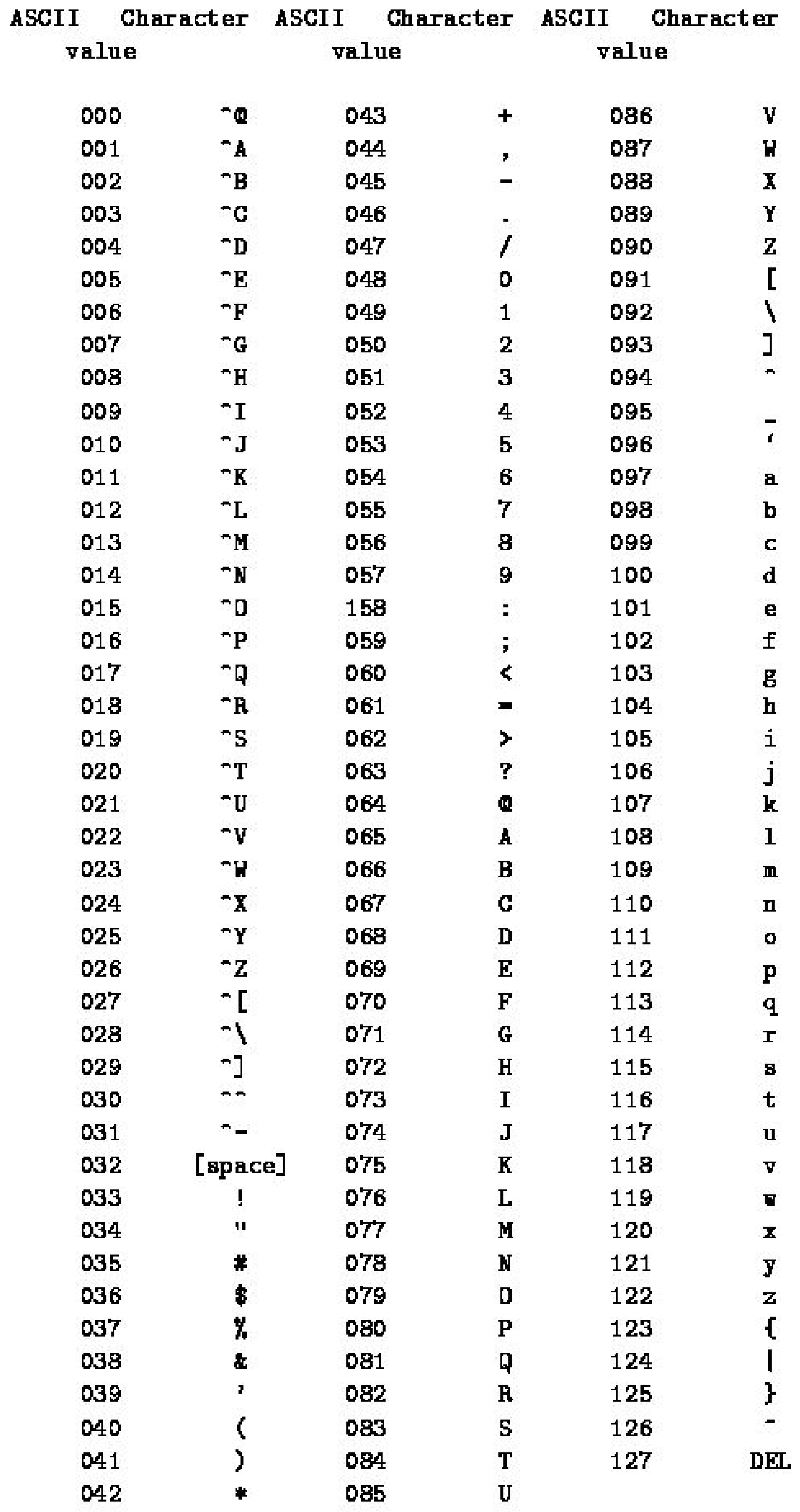
> min, min+1, max, max-1 → what about min-1 and max+1?

Additional boundaries may consider how data is represented in memory - maybe “hidden” boundaries when going from one byte to one word, from single to double.



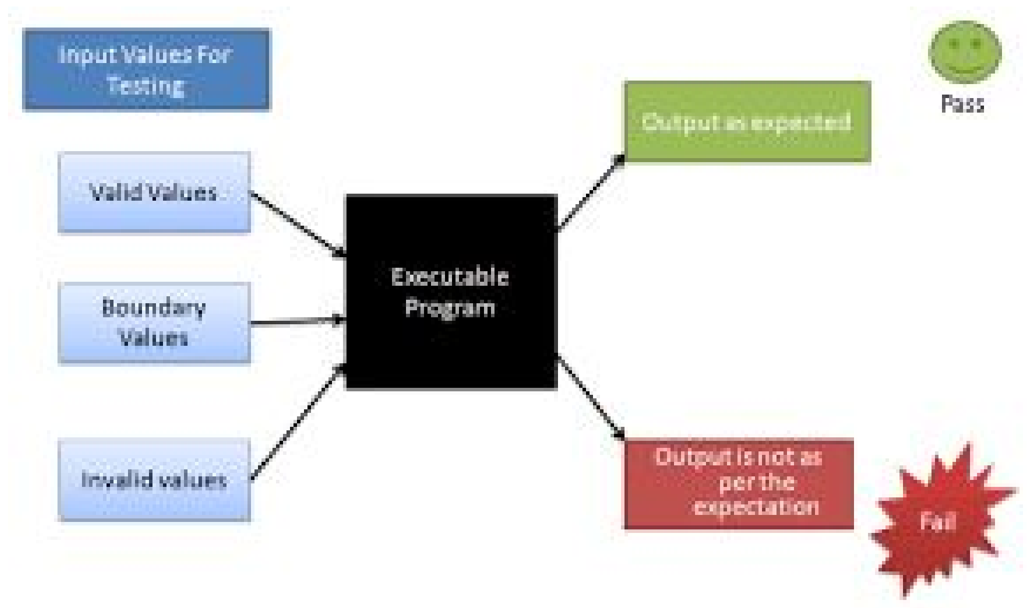
If the size of the data matters to the code, then there should be branches corresponding to the boundaries at, just above and just below each relevant size.

What are the boundaries for ascii?



ASCII value	Character	ASCII value	Character	ASCII value	Character
000	^@	043	+	086	V
001	^A	044	,	087	W
002	^B	045	-	088	X
003	^C	046	.	089	Y
004	^D	047	/	090	Z
005	^E	048	0	091	[
006	^F	049	1	092	\
007	^G	050	2	093	]
008	^H	051	3	094	^
009	^I	052	4	095	_
010	^J	053	5	096	‘
011	^K	054	6	097	a
012	^L	055	7	098	b
013	^M	056	8	099	c
014	^N	057	9	100	d
015	^D	158	:	101	e
016	^P	059	;	102	f
017	^Q	060	<	103	g
018	^R	061	=	104	h
019	^S	062	>	105	i
020	^T	063	?	106	j
021	^U	064	@	107	k
022	^V	065	A	108	l
023	^W	066	B	109	m
024	^X	067	C	110	n
025	^Y	068	D	111	o
026	^Z	069	E	112	p
027	^[	070	F	113	q
028	^\	071	G	114	r
029	^]	072	H	115	s
030	^^	073	I	116	t
031	^-	074	J	117	u
032	[space]	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	D	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

Table 4.2: ASCII Table



Are equivalence classes plus boundary analysis, including invalid classes and at/outside the boundaries, sufficient to cover all branches?

Need to make sure *all* inputs and outputs are accounted for, including returns from system and library calls.

```
if (random() is odd)
  { do something}
else
  { do something else }
```

In general, if we track coverage during execution of black box 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.

```
if (random() is odd)
  { do something}
else
  { do something else }
```

What tests would be needed to cover every statement?
What tests would be needed to cover every branch?

```
foo (int Y) {
    if (Y<=0) { Y = -Y; }
    while (Y>0) {
        do something;
        Y = Y-1;
    }
}
```

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

How can we record branch coverage?

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

Manually adding logging code would be very tedious for non-trivial programs, and might confuse debugging.

Better to use coverage tools (or compiler options) that automatically insert coverage-checking code at the 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.

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

May also count how many times each element executed.

There are research tools that generate test suites. Randoop automatically generates “feedback-directed” random tests. EvoSuite generates test suites that fulfill some criteria, such as as least one test case per branch

Let’s say we had such tools, why not use only white box testing, not black box?

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

Further, 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;
}
```

“Assignment T3: First Iteration” due November 17.
https://courseworks2.columbia.edu/courses/104335/assignments/490808

Implement your MVP (minimum viable product). Tell us what user stories you really implemented and what acceptance tests you used to check that the user stories really work. You must use build/package, unit testing, style checking, and static analysis bug finding.

“Assignment T4: Initial Demo” due November 23. Note this is the Monday before Thanksgiving.
https://courseworks2.columbia.edu/courses/104335/assignments/491014

We will hold very brief meetings now, 2-3 minutes, for teams to check in with their IA.
Breakout Room Assignments