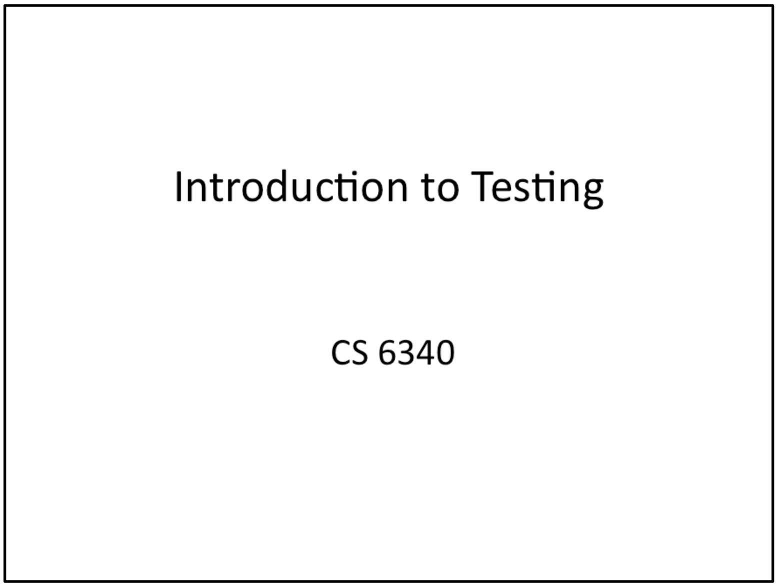


Introduction to Testing

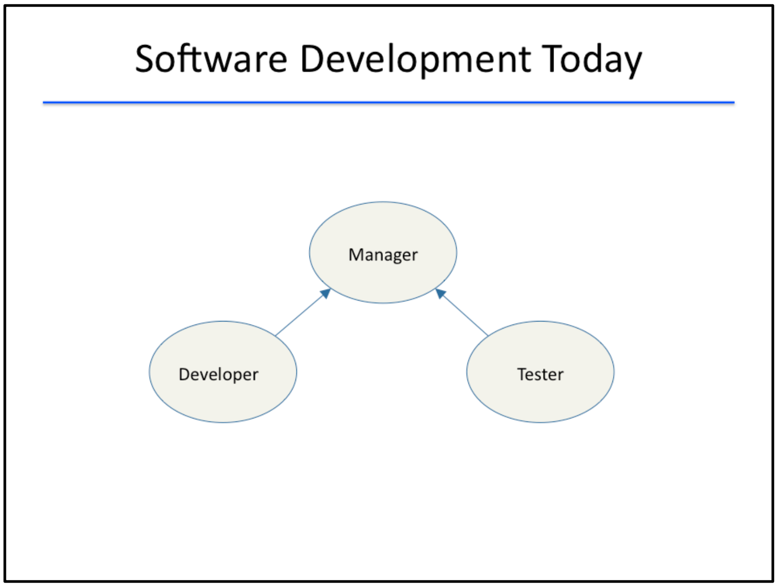
CS 6340

{HEADSHOT}

In the first lesson, you learned the basics of software analysis. However, analysis is only one half of this course's scope. The other half is software testing, the process of checking the correctness of a given piece of software.

In this lesson, you will learn the basics of software testing. By the end of this lesson, you will be able to:

- describe the relationship of software testing to software development,
- classify different testing methods along two key dimensions,
- identify the specifications that are required for testing software, and
- measure the quality of testing conducted on a given piece of software.



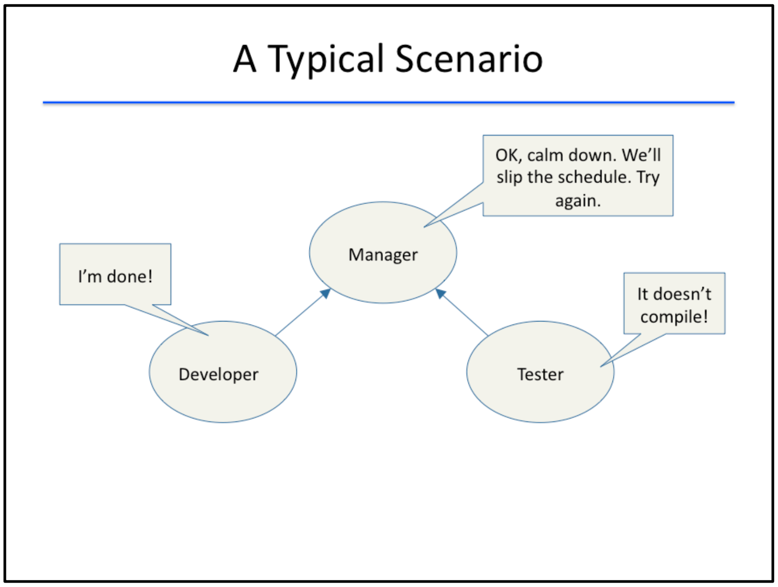
Software Development Today

Manager

Developer

Tester

Let’s start with an overview of how a typical software development team works. A team typically consists of at least one developer, at least one tester, and a manager to which both the developers and testers report.



A Typical Scenario

Manager

OK, calm down. We'll slip the schedule. Try again.

I'm done!

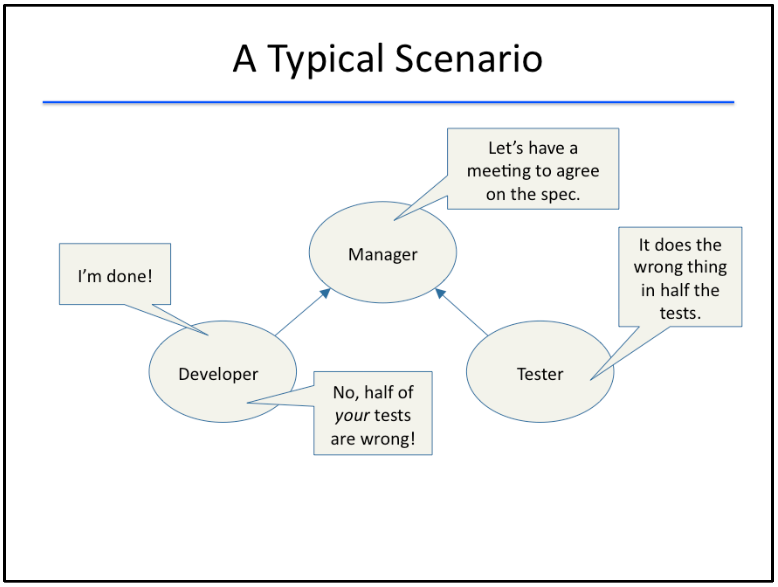
Developer

Tester

It doesn't compile!

The developer finishes writing some code for a project, and then sends it to the tester to make sure the code works.

The tester, however, is unable to even compile the code, and the manager has to push back the schedule so the problem can be fixed.



A Typical Scenario

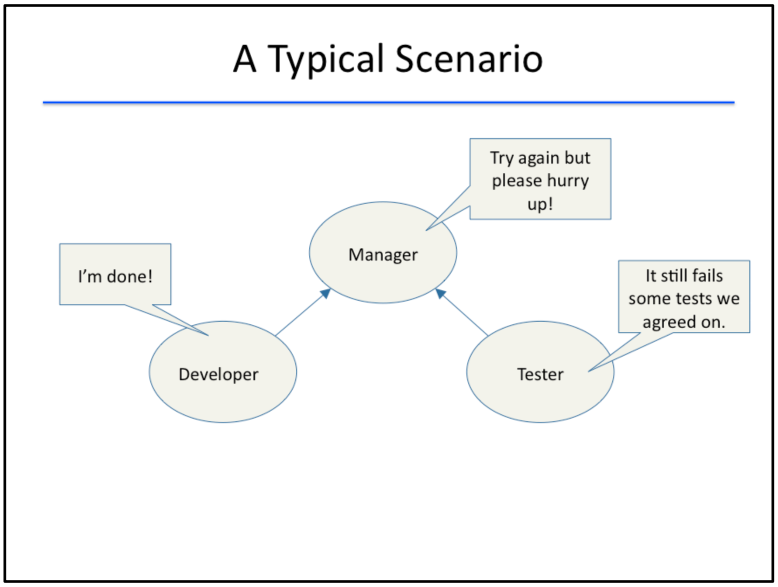
Let's have a meeting to agree on the spec.
Manager
I'm done!
It does the wrong thing in half the tests.
Developer
Tester
No, half of *your* tests are wrong!

The developer fixes the code and sends it to the tester again.

This time, the tester is able to compile the code, and runs a test suite against the compiled code, but finds that the code does the wrong thing in half the tests.

This time, the developer claims that the problem lies not in the code but in the test suite, and that half of the tests in the test suite must be wrong.

So the manager has to step in and make sure the developer and tester agree on the specifications for the program.



A Typical Scenario

Try again but please hurry up!

Manager

I'm done!

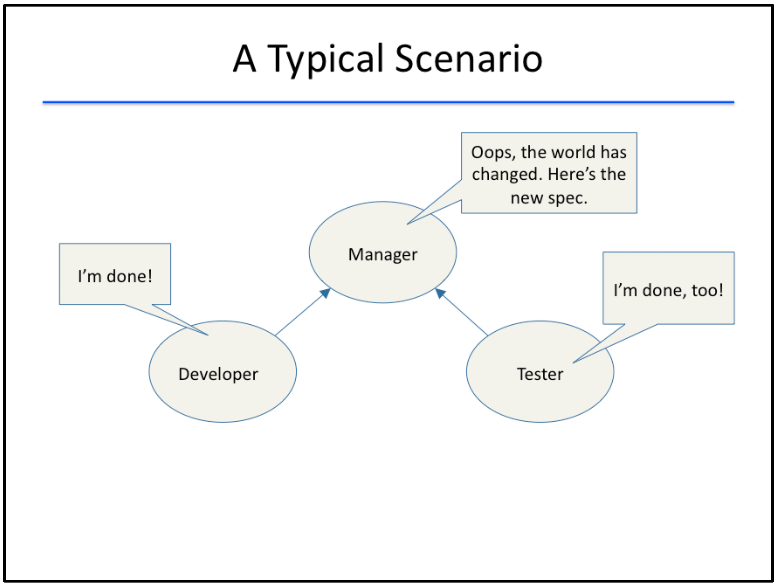
It still fails some tests we agreed on.

Developer

Tester

The developer fixes the code again according to the newly decided specifications, but the tester still finds problems with the code.

So the manager has to push back the project again.



A Typical Scenario

Oops, the world has changed. Here's the new spec.

Manager

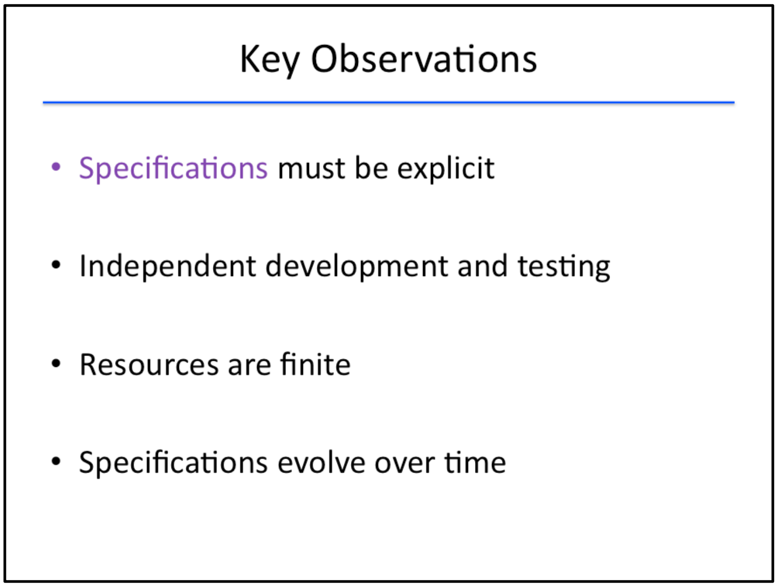
I'm done!

I'm done, too!

Developer

Tester

Finally, the developer finishes implementing the program to the satisfaction of the tester. But then the requirements for the program change, and the team is back to square one.



Key Observations

- Specifications must be explicit
- Independent development and testing
- Resources are finite
- Specifications evolve over time

We can make several key observations from this typical development scenario.

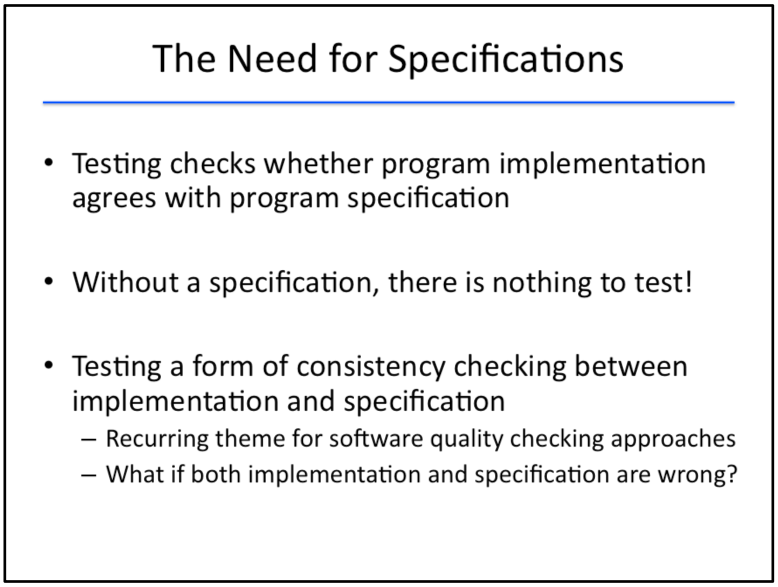
First, program specifications have to made explicit, or else there cannot be effective communication between those implementing code and those testing it.

Second, development and testing of a program are often done independently. This helps to ensure that errors in the code are caught.

Third, there are only finitely many resources available to development teams; not every bug can be found and caught.

Fourth, program specifications are not static; they evolve over time, and programming teams need to be able to adapt to these changes.

Let’s delve deeper into each of these observations.



The Need for Specifications

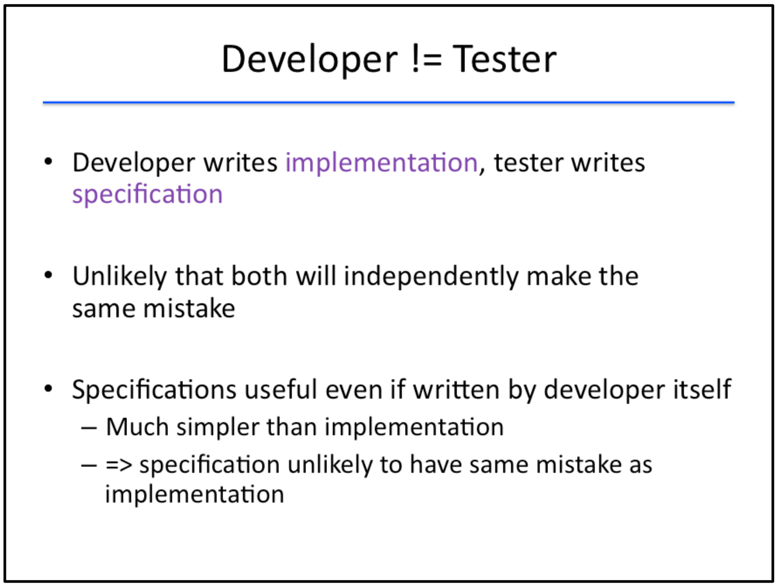
- Testing checks whether program implementation agrees with program specification
- Without a specification, there is nothing to test!
- Testing a form of consistency checking between implementation and specification
  - Recurring theme for software quality checking approaches
  - What if both implementation and specification are wrong?

The first observation from the scenario we saw was that specifications must be explicit. This is because the goal of testing is to check whether the implementation of the program agrees with a specification of the program. We will come to the topic of what specifications look like shortly.

But the main thing to note for now is that, without a specification, there is nothing to test! Testing, then, can be viewed as a form of consistency checking between the program's implementation and specification. This is a recurring theme for all software quality checking approaches including testing.

This points to a potential shortcoming of all these approaches including testing: both the implementation and specification could be wrong, and these approaches would not be able to notice the problem, as they would declare the two to be consistent.

This leads us to our second observation.



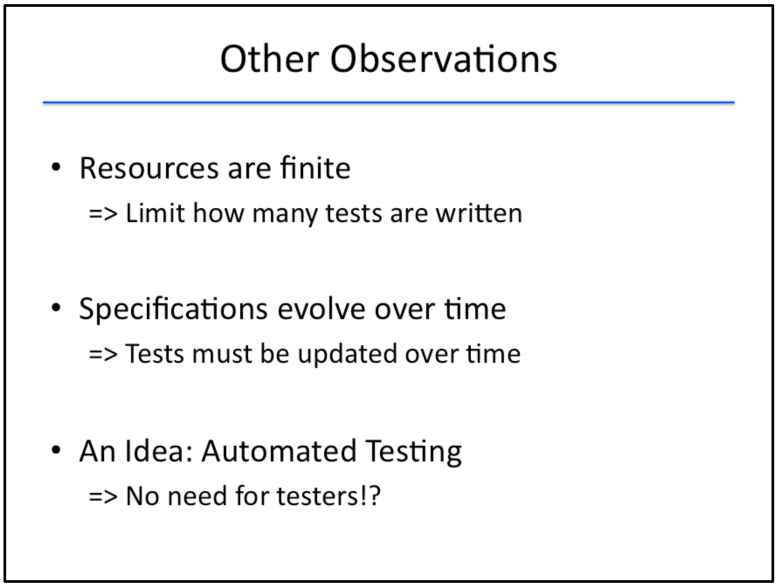
Developer != Tester

- Developer writes implementation, tester writes specification
- Unlikely that both will independently make the same mistake
- Specifications useful even if written by developer itself
  - Much simpler than implementation
  - => specification unlikely to have same mistake as implementation

The second observation from the scenario we saw was that the developer and tester were independent. The developer writes the program implementation, that is, what the program actually does, while the tester writes a program specification, that is, a facet of what the program is expected to do.

Since the two parties are independent, it is less likely that both will make the same mistake, that is, the developer will commit a bug in the program's implementation and the tester will affirm that same bug in the specification. It is due to this independence that consistency checking makes sense.

So we saw that testing is useless without specifications. We can also ask the converse question: are specifications useless without testers? That is, suppose you are a developer and there is no independent tester for the program you are developing. Then, does it make sense for you to write specifications? The answer is yes. This is because, even though the same person -- that is, the developer -- writes both the implementation and the specification, the specification artifact is typically much simpler than the implementation artifact. This is because each specification checks only one facet of the implementation. So the specification is still unlikely to contain the same bug as the implementation.



Other Observations

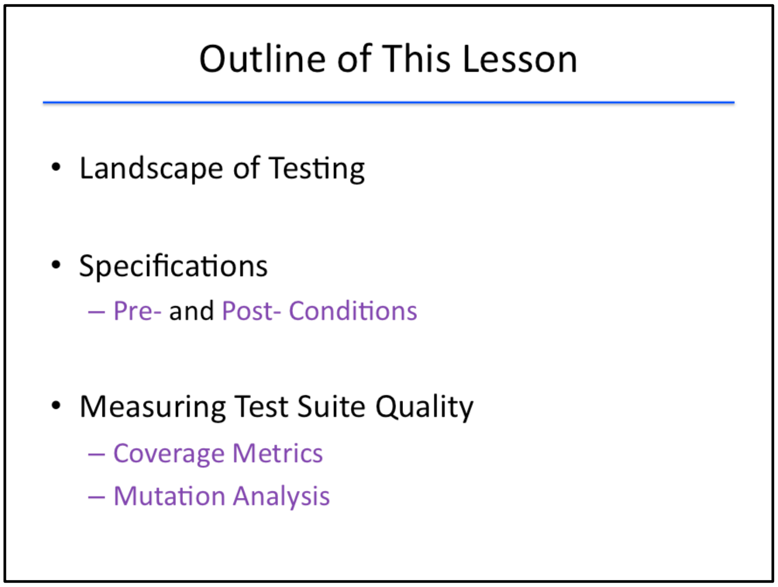
- Resources are finite
  => Limit how many tests are written
- Specifications evolve over time
  => Tests must be updated over time
- An Idea: Automated Testing
  => No need for testers!?

Additionally, the resources available to a development team are limited and must be prioritized. There is a finite number of tests that can be written and run for a given piece of code. The tests we write for a program will only be able to check certain facets of the program; they won't be able to check every possible use case.

And the specifications for a program change over time, so a given set of tests for a program will not necessarily remain valid for the lifetime of the software. Resources must be spent on updating test suites in order to reflect the new specifications.

Given all these observations---which we can summarize as the fact that testing is a necessary yet ongoing, resource-intensive process---wouldn't it be nice to be able to *automate* the process of writing and running tests? In the extreme case, we'd never need another QA department ever again!

While we are certainly not at the point of obsoleting human testing, this idea of automated testing is attractive. In future lessons, we will explore some of the basics of automated testing and introduce you to some of the automated testing techniques currently used in the industry today.



Outline of This Lesson

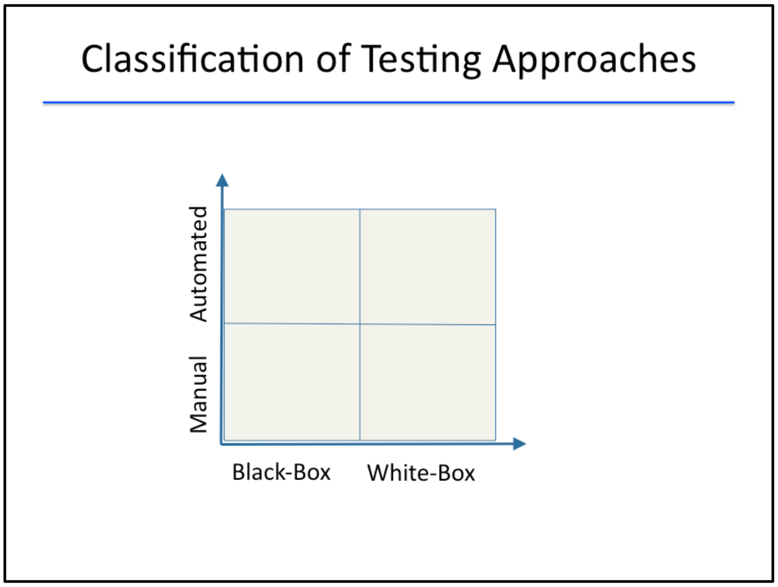
- Landscape of Testing
- Specifications
  - Pre- and Post- Conditions
- Measuring Test Suite Quality
  - Coverage Metrics
  - Mutation Analysis

Here is a road map for the remainder of this lesson.

We will begin by surveying the landscape of testing paradigms, comparing and contrasting their costs and benefits. By the end of this section, you should have an understanding of what characterizes different testing strategies and should be able to select an appropriate strategy given a set of priorities and constraints.

Next, we will look at specifications in more detail. In particular, you will see how to use pre-conditions and post-conditions to specify the behavior (or at least certain facets of the behavior) of functions.

Finally, we will wrap up the lesson by studying two methods of quantifying the quality of a given set of tests, or test suite. These methods will allow you to determine whether a test suite for a program is doing its job or whether it needs to be augmented with more or more diverse tests.



Classification of Testing Approaches

Approaches to testing software can be classified according to two orthogonal axes: manual vs. automated and black-box vs. white-box.

The vertical axis describes the amount of human participation in the testing process: testing that requires more human direction falls closer to the manual side of the axis, while testing that is performed primarily without human direction falls closer to the automated side of the axis.

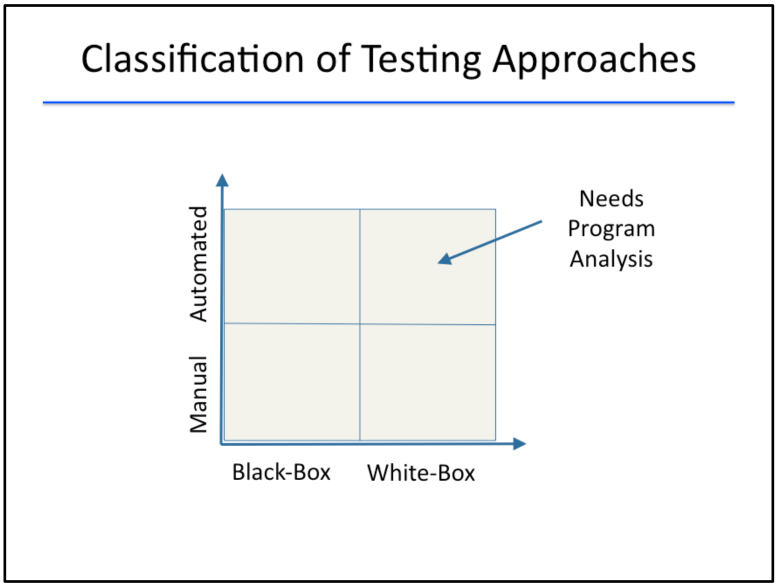
The horizontal axis describes the amount of access the testing apparatus has to the tested program's source code.

Black-box testing refers to testing where the tester can see nothing about the tested program's internal mechanisms: as though the program is contained inside an opaque box. The tester can only issue inputs to the program, observe the program's outputs, and determine whether the observed outputs meet the specifications required of the program.

White-box testing refers to testing in which the internal details of the program being tested are fully available to the tester. The tester can use these internal details to perform a more precise analysis of the tested program and uncover inputs that are more likely to trigger buggy behavior.

Testing approaches need not be strictly black-box or white-box; some internal details may be available to the tester while others are hidden. These sorts of testing approaches are called "gray-box" approaches.

Similarly, testing approaches need not be fully manual or fully automatic. It is better to think of these axes as continua instead of as discrete categories.



Classification of Testing Approaches

Needs Program Analysis

Automated

Manual

Black-Box

White-Box

Let’s look at some specific examples of testing approaches and see where they fall along the two axes.

One testing approach is for the tester to tinker with observational behavior of a program: for example, exercising different GUI events of an Android app. This sort of testing would fall somewhere near the bottom-left of this diagram because the tester is only issuing commands to and observing outputs from the program under testing, and this is done on a manual basis.

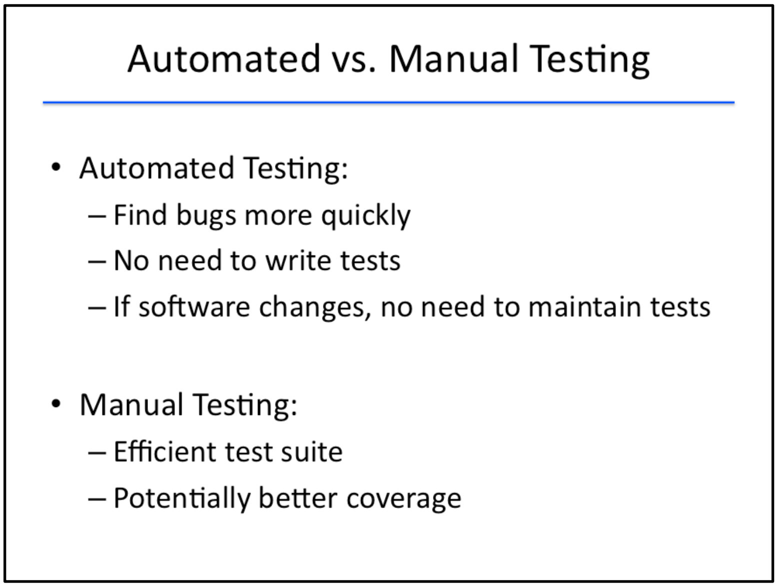
Now, if we were to modify this testing approach so that the tester looks at the source code in order to determine what possible routes there are through the app’s GUI, then we have taken the original approach and moved it rightward to the white-box side of the diagram.

You are probably familiar with both of these types of testing. Testing approaches we will learn in this course primarily will focus on the top two quadrants, that is, how to leverage modern tools and techniques to automate testing.

For example, instead of manually activating GUI events for the Android app, we might use an automated approach such as a fuzzer to automatically issue tap commands to random coordinates of the smartphone. This takes the original black-box approach and moves it upward to the automated side of the diagram. This is not a very sophisticated approach, but it has many advantages that we will see in the next lesson.

We might also take approaches such as feedback-directed random testing (issuing random commands that change in response to feedback issued by the GUI), perform symbolic execution that needs to inspect the source code in order to test effectively (i.e. perform static analysis), or even monitor the code as it is being tested (i.e. perform dynamic analysis) in order to discover future tests intelligently. The approaches take us to the top-right quadrant of the diagram, in which we are performing automated, white-box testing of a program. We’ve already alluded to some of these approaches earlier in the first lesson; we will discuss them later in the course.

Next let’s look at pros and cons of different approaches along each of these two dimensions.



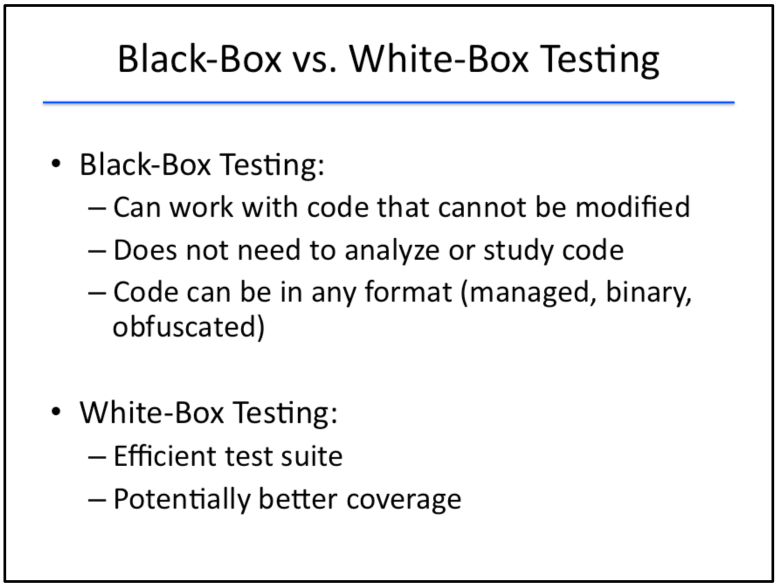
Automated vs. Manual Testing

- Automated Testing:
  - Find bugs more quickly
  - No need to write tests
  - If software changes, no need to maintain tests
- Manual Testing:
  - Efficient test suite
  - Potentially better coverage

One of the advantages of automated testing over manual testing is that we can potentially spot bugs more quickly through the speed advantage of a computer over a human in issuing inputs and checking outputs. Additionally, there is no need to write tests: they are generated by the computer itself. Moreover, if the software changes, there is no need to update the tests by hand, as the computer will generate new tests relevant to the updated software.

On the other hand, an advantage of manual testing is that humans are potentially better able to select an efficient set of tests: computer-generated test suites can be rather bloated. Additionally, humans can potentially construct a test suite with better code coverage than a computer program could, though this is not guaranteed.

The ideal approach will often lie in the combination of automated and manual approaches: a semi-automated approach. An example of such an approach is where a human specifies the format or the grammar of valid inputs to a program, so that the automated testing that follows does not waste resources generating invalid inputs that do not exercise any interesting functionality of the program.



Black-Box vs. White-Box Testing

- Black-Box Testing:
  - Can work with code that cannot be modified
  - Does not need to analyze or study code
  - Code can be in any format (managed, binary, obfuscated)
- White-Box Testing:
  - Efficient test suite
  - Potentially better coverage

Black-box testing has many advantages over white-box testing. First, it does not require modifying the code, that is, introducing probes into the code. This is a big advantage because in many practical cases, the tester does not have the liberty to modify code.

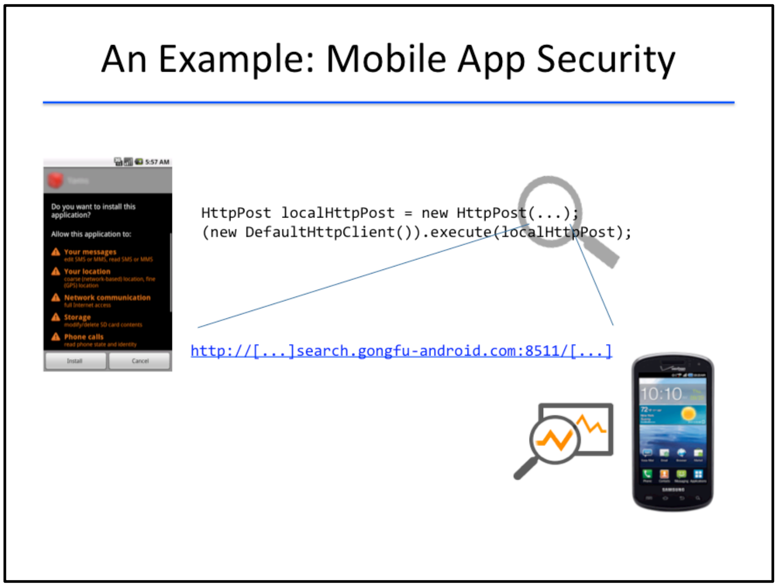
For instance, consider an Android application. Its code comprises not only the code of the application itself but also parts of the Android framework that the application is built upon. So a tester would need to modify not only the application's code but also the entire framework's code.

Additionally, black-box testing does not need to study the code to be tested. The problem with analyzing this code is that it might be too low-level of an analysis: the tool might get lost in details of the program and lose sight of the bigger picture that observing inputs and outputs provides.

Moreover, black-box testing can be performed on any format of code, whether it is managed code, binary code, obfuscated code, etc.

The advantages of white-box testing over black-box testing are similar to those of manual over automated testing: the tester is potentially able to construct a more efficient suite of test cases with potentially better coverage than black-box testing could.

Both kinds of approaches are useful, and typically a combination is needed. Let's take a look at a concrete example next that illustrates black-box and white-box testing.



An Example: Mobile App Security

5:57 AM

Do you want to install this application?

Allow this application to:

- Your messages
  edit SMS or MMS, read SMS or MMS
- Your location
  coarse (network-based) location, fine (GPS) location
- Network communication
  full Internet access
- Storage
  modify/delete SD card contents
- Phone calls
  read phone state and identity

Install
Cancel

```
HttpPost localHttpPost = new HttpPost(...);
(new DefaultHttpClient()).execute(localHttpPost);
```

http://[...]search.gongfu-android.com:8511/[...]

Let’s compare and contrast black-box and white-box testing for the problem of determining whether an Android app is malicious.

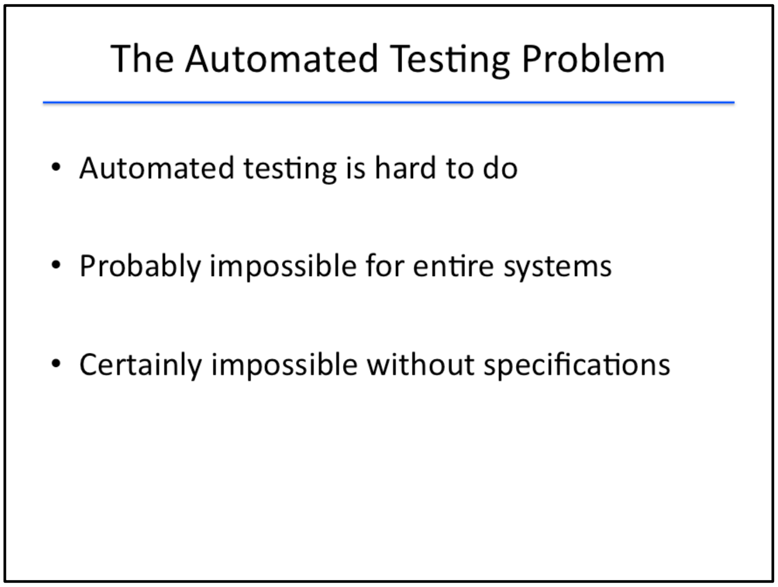
Consider the DroidKungFu malware, which was found to be in circulation on eight 3rd-party app stores based out of China around 2011. Prior to installation, this app asks for the following permissions. [Permissions on the left of the slide appear]

Once installed, the app attempts to collect sensitive information from the compromised device, and reports it to remote command & control servers at multiple web locations such as this one.

Black-box testing would detect this malware by merely starting the app and monitoring the network activity of the phone, thereby capturing the attempt to connect to this suspicious web location.

White-box testing, on the other hand, would involve inspecting the source or binary code of the app, instead of observing the app’s input-output behavior in the case of black-box testing. This inspection in turn would reveal this call in the code to connect to this suspicious web location.

Notice that both the approaches detect the same malicious behavior but they do so in fundamentally different ways. Of course, either of these approaches could fail to detect this malicious behavior if they are not careful enough; a combined approach would reduce the chance of such failure.



The Automated Testing Problem

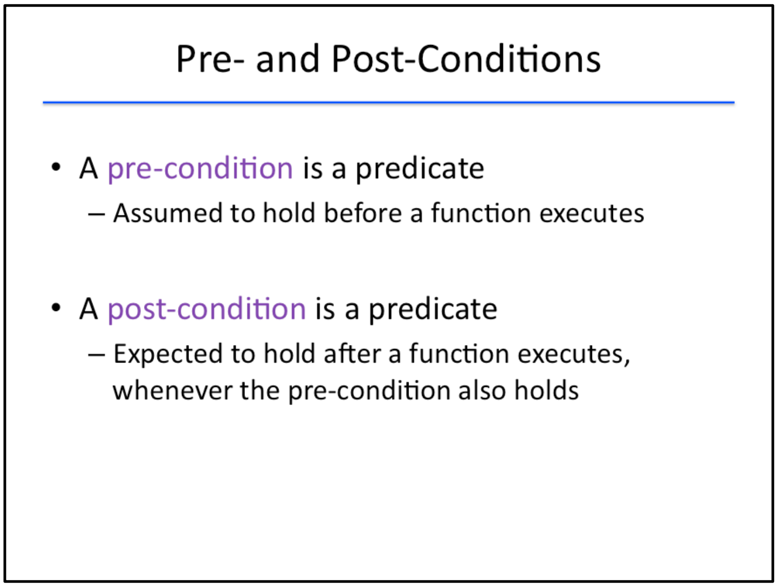
- Automated testing is hard to do
- Probably impossible for entire systems
- Certainly impossible without specifications

Despite the attractiveness of automating away all of our testing, there are several constraints that prevent us from making testing entirely automatic.

As we will see, testing is a hard enough problem even for a small piece of code. The number of pathways through a program increases exponentially with the number of branch-points in the program: just 30 if-else statements yield over one billion possible routes that need to be tested to verify that the program works in all conditions. And if a program has a loop, there could potentially be an infinite number of routes through the code, in which case it becomes impossible to test the code under all possible conditions.

Moving beyond the scope of a single file of code to an entire system, the problem of testing quickly becomes intractable. The best we can hope to do in many cases is separate code into small components, each of which is tested separately.

And if we don't have a specification for our program, then no testing can be done at all, let alone automated testing! So let's start by looking at how to define the specifications for a program.



Pre- and Post-Conditions

- A pre-condition is a predicate
  - Assumed to hold before a function executes
- A post-condition is a predicate
  - Expected to hold after a function executes, whenever the pre-condition also holds

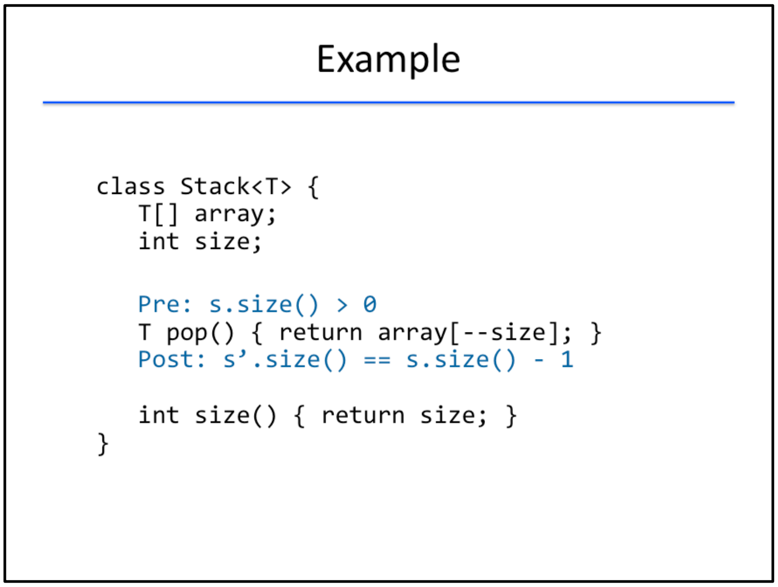
Let’s look at very general specification mechanisms called pre- and post-conditions.

A pre-condition is a predicate that is assumed to hold before the execution of some function, and a post-condition is a predicate that is expected to hold after the execution of a function whenever the pre-condition holds.

Pre- and post-conditions can be considered as special kinds of assertions, which we saw in the first lesson.

One use of pre-conditions is to ensure that a function does not operate in an undefined way on inputs it was not designed to handle.

Similarly, a use of post-conditions is to ensure that a function’s output matches its specification: a function that squares a real number should not output a negative number, for example.



Example

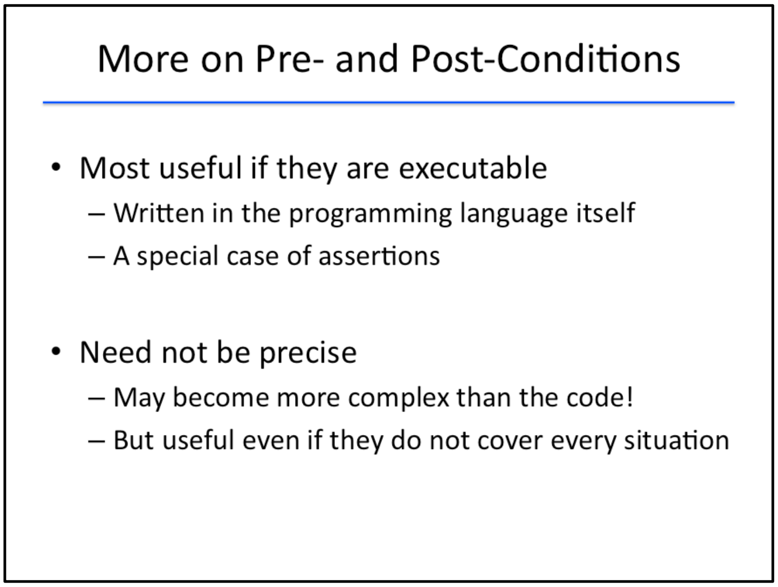
```
class Stack<T> {
   T[] array;
   int size;

   Pre: s.size() > 0
   T pop() { return array[--size]; }
   Post: s’.size() == s.size() - 1

   int size() { return size; }
}
```

In this example code we see a pre-condition and a post-condition for the function pop() in the class Stack.

An assumption is made going into the function that the stack has at least one element (otherwise pop() is undefined; indeed, in Java trying to execute this pop() method with an empty Stack object would throw an exception upon trying to access the element at index -1 of array).

The post-condition asserts that the size of the Stack object after the pop (s’.size()) should be exactly one smaller than it was beforehand (s.size()). This post-condition also documents a change in the Stack object’s state that outside code can rely upon whenever it calls the pop function.

Remember that pre- and post-conditions often only partially capture the specifications of a function. For instance, the above post-condition says nothing about remaining N - 1 elements on stack (e.g., whether they are in the same order as before, or even whether they are in fact the same objects as before!). Implicitly, this lack of mention is interpreted as saying that “nothing else changes” about the state of the program. These sorts of implied conditions are called frame conditions.



More on Pre- and Post-Conditions

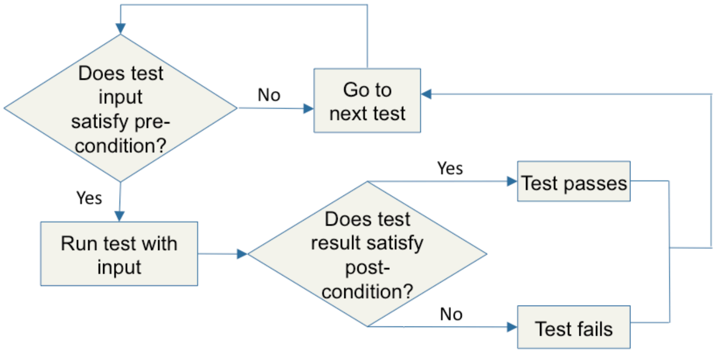
- Most useful if they are executable
  - Written in the programming language itself
  - A special case of assertions
- Need not be precise
  - May become more complex than the code!
  - But useful even if they do not cover every situation

Pre- and post-conditions are most useful to developers and testers if they are executable. Indeed, we can write the conditions into the program itself in the same language, either using a testing framework such as JUnit or built-in functionality such as an assert statement.

Pre- and post-conditions also need not be precise statements of the required conditions on the input and output of each function. Recall earlier that the problem of testing quickly becomes intractable because of the number of possible routes execution flow through a program can contain.

Similarly, pre- and post-conditions that perfectly describe the input and output requirements for a function may be extremely complex, perhaps even more than the code it is specifying. As such, these conditions often only check certain facets of the input and output, trading precision for tractability.

Using Pre- and Post-Conditions


Does test input satisfy pre-condition?
No
Go to next test
Yes
Run test with input
Does test result satisfy post-condition?
Yes
Test passes
No
Test fails

Doesn't help write tests, but helps run them

The process of using pre- and post-conditions in testing is straightforward.

To perform a test, we first check that the test input satisfies the pre-condition. [Left Diamond appears]

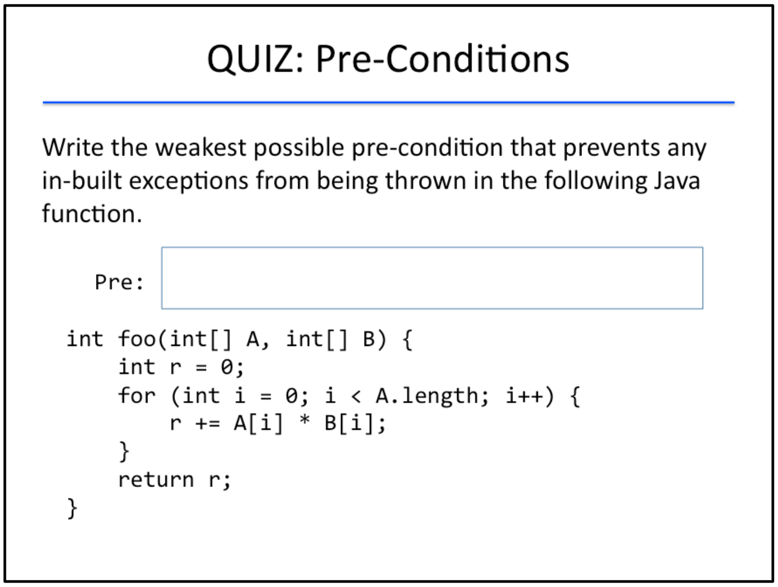
If it does not, then we skip the test and go to the next one. ["No" arrow, "Go to next test" box, and rectangular arrow out of that box appears]

If it does, then run the test with that input, ["Yes" arrow and "Run test with input" box appears] and then check that the output of the test satisfies the post-condition. [Arrow out of "Run test with input" box and Right Diamond appears]

If it does, then the test passes ["Yes" arrow and "Test passes" box appears] Otherwise, the test fails ["No" arrow and "Test fails" box appears] In both of these cases, we then proceed to the next test. [All the remaining arrows appear]

In this way, we check that any input satisfying the pre-condition yields a result satisfying the post-condition.

While this framework doesn't help us generate the tests, it does help significantly with automating testing runs.



QUIZ: Pre-Conditions

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

Pre: □

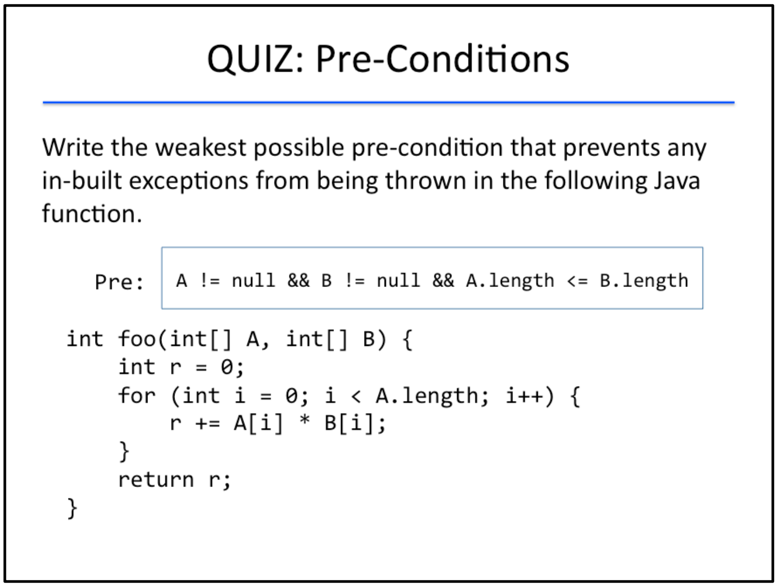
```
int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

{QUIZ SLIDE}

To check your understanding of pre-conditions, let’s take a look at this Java function.

For this quiz, write the weakest possible pre-condition that prevents any built-in exceptions (such as NullPointerException and ArrayIndexOutOfBoundsException) from being thrown during any execution of the program.

Write your answer as a Java boolean expression in the text box provided. (No need to write an assert statement here: just a boolean expression.)



QUIZ: Pre-Conditions

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

Pre: `A != null && B != null && A.length <= B.length`

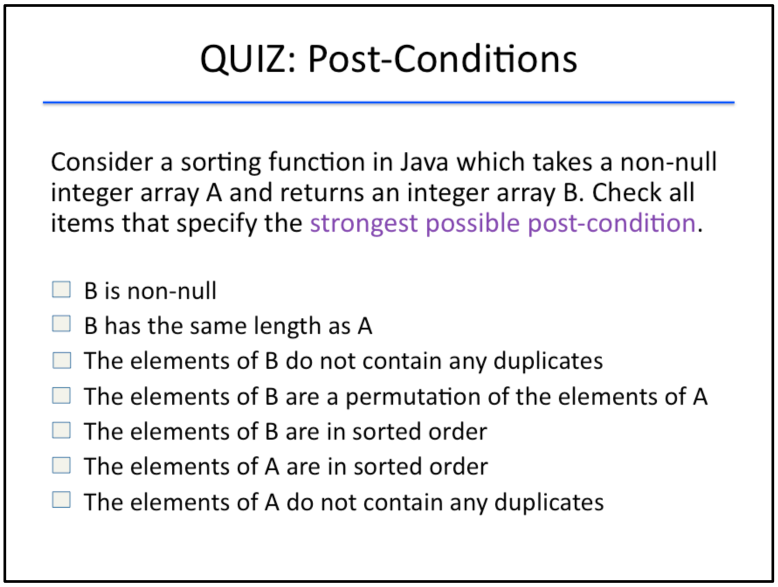
```
int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

{SOLUTION SLIDE}

For this function, our pre-condition needs to assert something about the input arrays A and B. Because we dereference both A and B, it is essential that they not point to a null array. So A != null && B != null should be included in the pre-condition.

Additionally, observe that for every cell we access of the array A, we also access the corresponding cell of the array B. Since this happens for every cell in A, we must also require the length of B to be at least the length of A or else we will run into an out-of-bounds exception. So we add on && A.length <= B.length to our pre-condition.

These are the weakest requirements we need in order for this function to execute correctly.



QUIZ: Post-Conditions

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the strongest possible post-condition.

□ B is non-null
□ B has the same length as A
□ The elements of B do not contain any duplicates
□ The elements of B are a permutation of the elements of A
□ The elements of B are in sorted order
□ The elements of A are in sorted order
□ The elements of A do not contain any duplicates

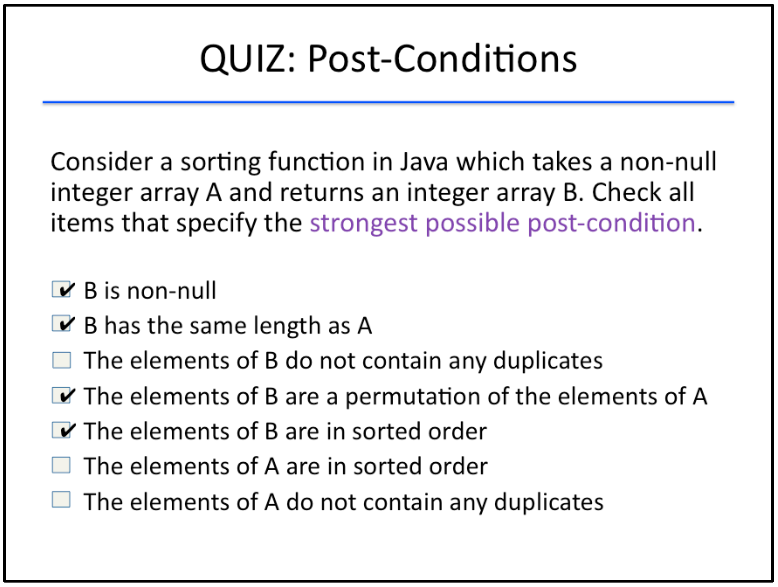
{QUIZ SLIDE}

Now, to check your understanding of postconditions, consider the following quiz.

Given a sorting function in Java which takes a non-null integer array A, puts them in sorted order in an integer array B, and then returns B, which of the following statements must be true in order to form the strongest possible postcondition for the function that does not improperly reject a correctly sorted array?

- B is non-null
- B has the same length as A
- The elements of B do not contain any duplicates
- The elements of B are a permutation of the elements of A
- The elements of B are in sorted order
- The elements of A are in sorted order
- The elements of A do not contain any duplicates

Check all the statements that apply.



QUIZ: Post-Conditions

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the strongest possible post-condition.

☑ B is non-null
☑ B has the same length as A
□ The elements of B do not contain any duplicates
☑ The elements of B are a permutation of the elements of A
☑ The elements of B are in sorted order
□ The elements of A are in sorted order
□ The elements of A do not contain any duplicates

{SOLUTION SLIDE}

Must B be non-null? Yes (since we assumed the sorting function was passed a non-null array in the first place).

Must B have the same length as A? Yes, sorting the elements of an array does not change the number of elements in the array.

Must B contain no duplicates? No, this is not required of a sorted array.

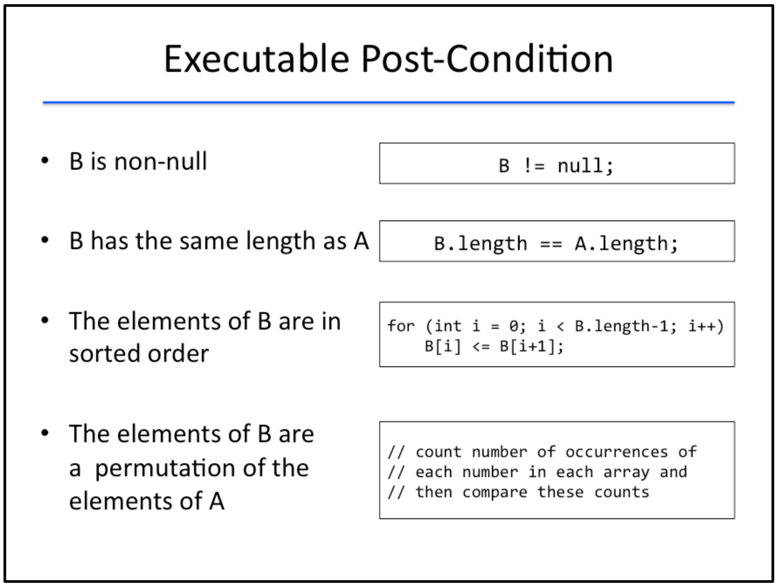
Must B be a permutation of A? Yes, B should consist of the same elements as A, just in a (possibly) different order.

Must B be in sorted order? Yes, of course (or else the function didn't do its job!).

Must A be in sorted order? No, this needn't be required, since we are not sorting A in place.

Must A contain no duplicates? No, this isn't required either, since sorting functions should be able to handle duplicate elements.

In fact, these four conditions make up the entirety of the specification for the output of a sorting function: there is no stronger condition that can be required without rejecting a properly sorted array.



Executable Post-Condition

- B is non-null

```
B != null;
```

- B has the same length as A

```
B.length == A.length;
```

- The elements of B are in sorted order

```
for (int i = 0; i < B.length-1; i++)
    B[i] <= B[i+1];
```

- The elements of B are a permutation of the elements of A

```
// count number of occurrences of
// each number in each array and
// then compare these counts
```

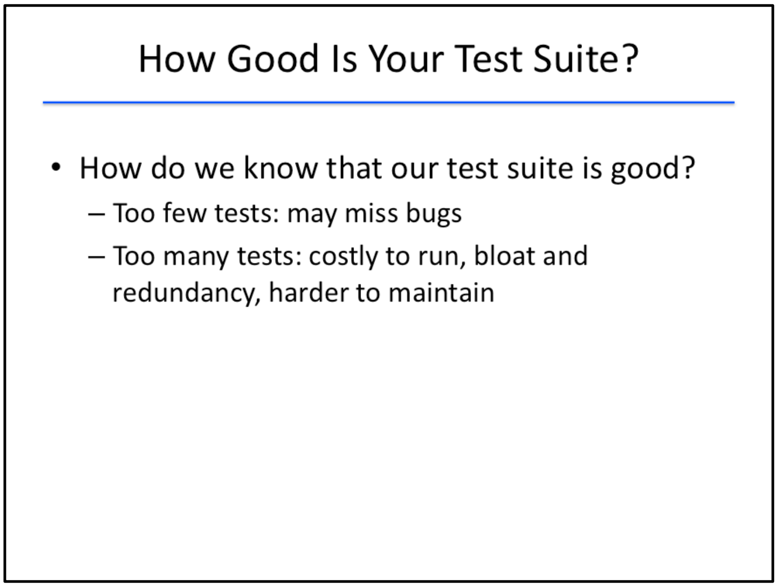
What would the sorting post-condition look like if we wrote it in executable code?

The first part of the post-condition is that B be non-null. This is easy enough to implement by adding an assertion that B not equal the null pointer.

The second part of the post-condition, that B and A have the same length, is similarly easy to implement, by adding an assertion that B.length == A.length.

The third part of the post-condition, that the elements of B are in sorted order, is a bit more complex. Here, we would need to check that B[i] <= B[i+1] for all i between 0 and B.length - 2 (or >=, if we were doing a descending sort).

The last part of the post-condition is the most complex to implement. One strategy might be to count the number of occurrences of each element in each array and then check that these counts are the same for each array. We'll leave it as an exercise for you to implement this yourself if you like.



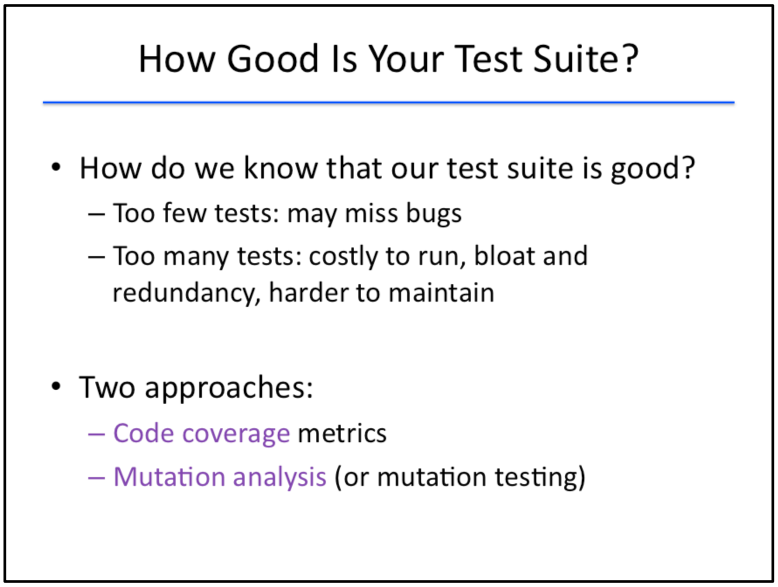
How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain

We have just developed one way of writing a program's specifications. Now, suppose we've developed a suite of test cases that check whether the program meets these specifications. Let's stop for a moment and think about the quality of our tests.

Have we included enough tests? Having too few tests is a bigger problem than having too many tests. If we have too few tests, we might miss regressions that occur as the code evolves; that is, a bug might not cause any of those few tests to fail, and thereby elude detection.

Have we included too many tests? If we have too many tests, running all of them before each code commit, or even nightly, could get expensive. Just like the problem of code bloat, we might run into the problem of test suite bloat, with lots of redundant tests. More tests are also harder to maintain and keep up-to-date than fewer tests.



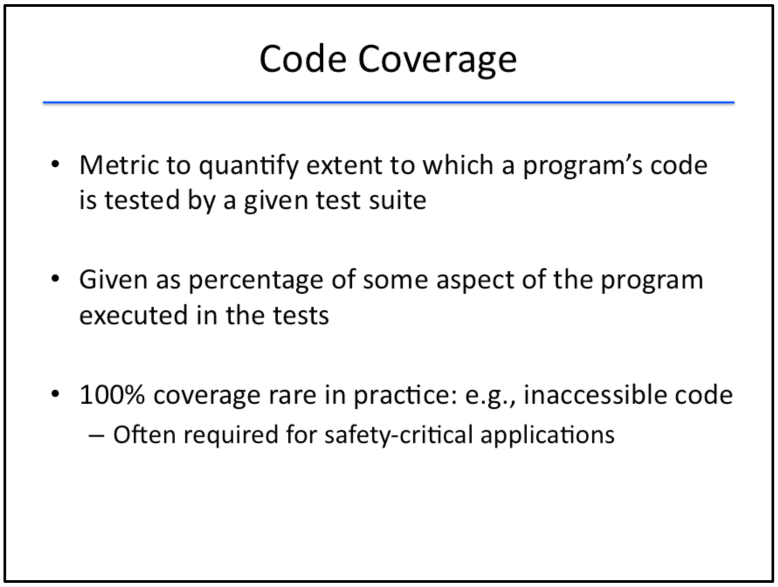
How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
- Two approaches:
  - Code coverage metrics
  - Mutation analysis (or mutation testing)

We will discuss two approaches to systematically quantify how good our testing suite is.

One approach to measuring the quality of our test suite is to use code coverage metrics. For example, we can check whether each statement of code has been executed at least once over the course of all tests, whether every possible route through the code has been taken, and so forth.

A second approach we can take to measure our test suite's quality is to use mutation analysis. In this approach, we randomly "mutate" the program under testing in various ways and then run the same tests on the mutant code as are used on the original code. If no tests fail on the mutant code, there is the implication that the testing suite may not be strong enough to distinguish correct from incorrect code.



Code Coverage

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as percentage of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., inaccessible code
  - Often required for safety-critical applications

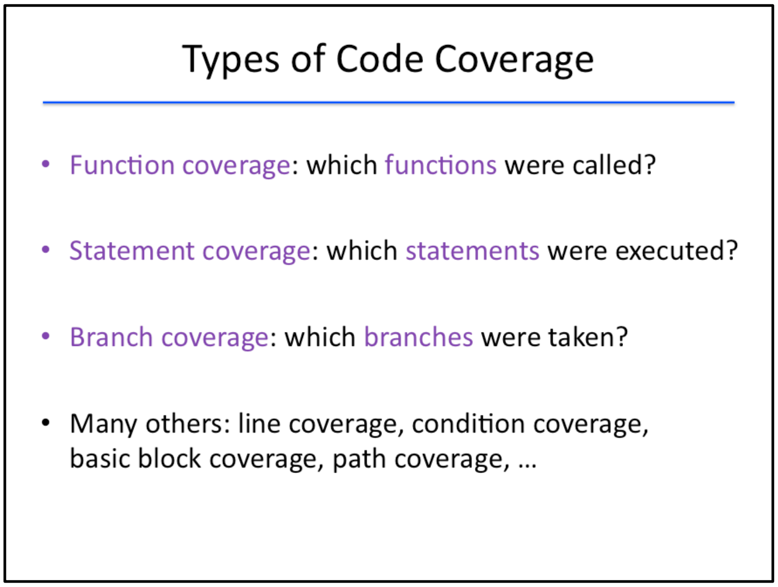
Let's study the first approach, code coverage, in some more detail.

Code coverage is a metric to quantify the extent to which a program's code is tested by a given test suite.

Code coverage is given as a percentage from 0 to 100 percent, of some aspect of the program that was executed in the tests. We will shortly see common examples of these aspects.

Higher code coverage is generally indicative of a better test suite and a better tested program. In practice, however, it's rare to see perfect or near-perfect coverage for a given program on a test suite. This often occurs because large systems have inaccessible or "dead" code.

However, some safety-critical applications (such as in airline navigation or nuclear weaponry) are often required to demonstrate perfect code coverage under some metric.



Types of Code Coverage

- Function coverage: which functions were called?
- Statement coverage: which statements were executed?
- Branch coverage: which branches were taken?
- Many others: line coverage, condition coverage, basic block coverage, path coverage, …

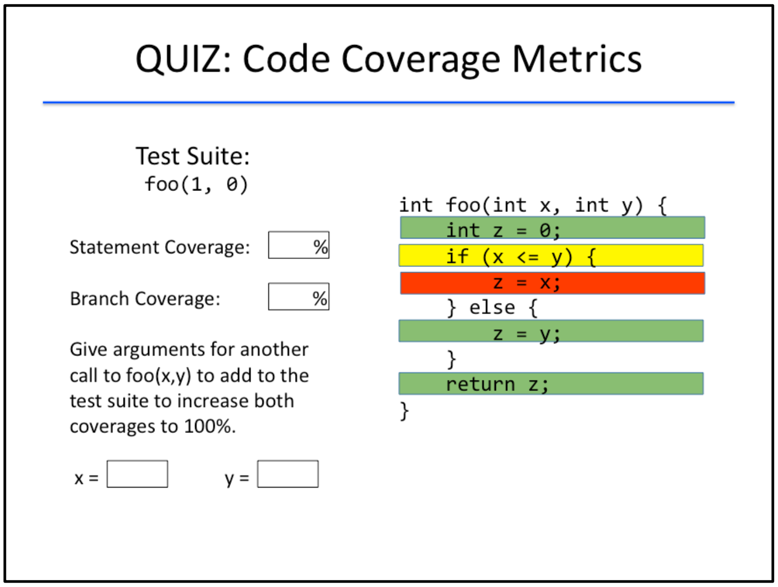
There are various aspects of programs that are commonly used to measure code coverage.

Function coverage measures the number of functions called out of the total number of functions in the program.

Statement coverage measures the number of statements that are executed out of all statements in a program.

Branch coverage measures the fraction of branches of each control structure that were taken (for example, only taking the “true” path of if-statements would result in at most 50% branch coverage).

And there are many others, such as line coverage, condition coverage, basic block coverage, and path coverage.



QUIZ: Code Coverage Metrics

Test Suite:

```
foo(1, 0)
```

Statement Coverage: □ %

Branch Coverage: □ %

Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.

x = □ y = □

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

{QUIZ SLIDE}

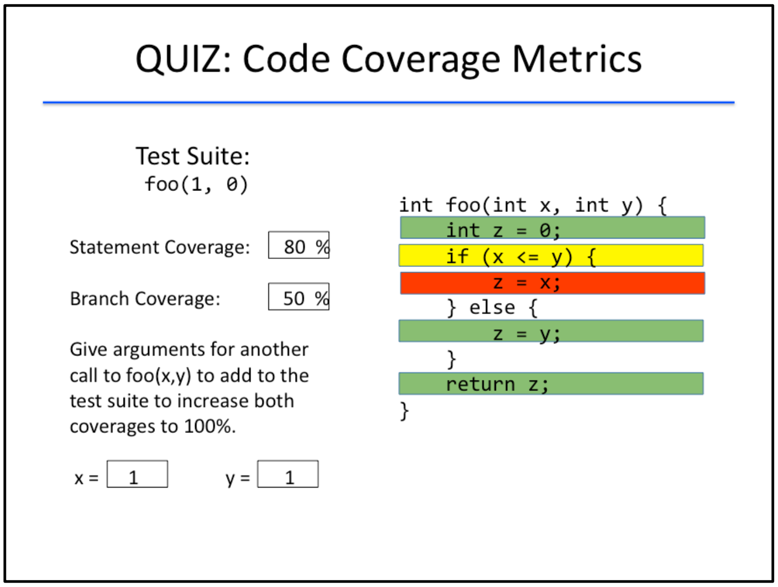
Let's look at coverage metrics for a small test suite.

In the code snippet to the right, we will highlighted each statement according to its coverage:

- Green will mean that the statement has been executed; for a control statement, this requires that the boolean statement controlling the flow evaluate to both true and false at some point during the suite.
- Yellow will mean that the line is a control statement where the controlling boolean expression has only been evaluated to either true or false but not both during the set of tests.
- Red will mean that the statement has not been executed in any test in the suite.

The first test we'll add to our suite is to execute foo(1,0). During this test, the statement int z = 0 is executed, so the statement is highlighted green. Then the boolean expression x <= y is evaluated to false, so we highlight the if-statement yellow and skip past z = x, leaving it red. We execute the statement z = y inside the else-block, so we highlight it green, and finally we execute the return statement, so we also highlight it green.

Now it's your turn. For this very small test suite, compute both the statement coverage and the branch coverage. Then, suppose we want to add a new function call to our test suite. Pick arguments for x and y that we can pass to foo in order to raise both of these coverage metrics to 100%.



QUIZ: Code Coverage Metrics

Test Suite:
`foo(1, 0)`

Statement Coverage: 80 %

Branch Coverage: 50 %

Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.

x = 1   y = 1

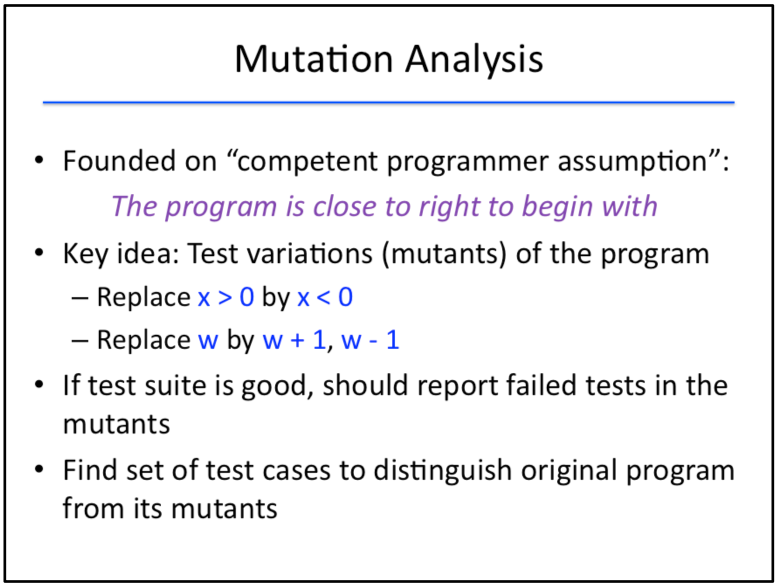
```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

{SOLUTION SLIDE}

Out of the five executable statements in this function, four are executed when we call foo(1,0), giving a statement coverage metric of 80%.

And, since the if-statement's boolean expression only evaluates to false but not true when we call foo(1,0), we've only covered 50% of the possible branches in this code.

In order to increase these metrics to 100%, we need to make sure that x <= y evaluates to true. Therefore, picking any set of arguments with x <= y, for example, x and y both 1, will ensure that the "true" branch of the if-statement is followed and that the statement z = x is executed.



Mutation Analysis

- Founded on “competent programmer assumption”:

  The program is close to right to begin with
- Key idea: Test variations (mutants) of the program
  - Replace x > 0 by x < 0
  - Replace w by w + 1, w - 1
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

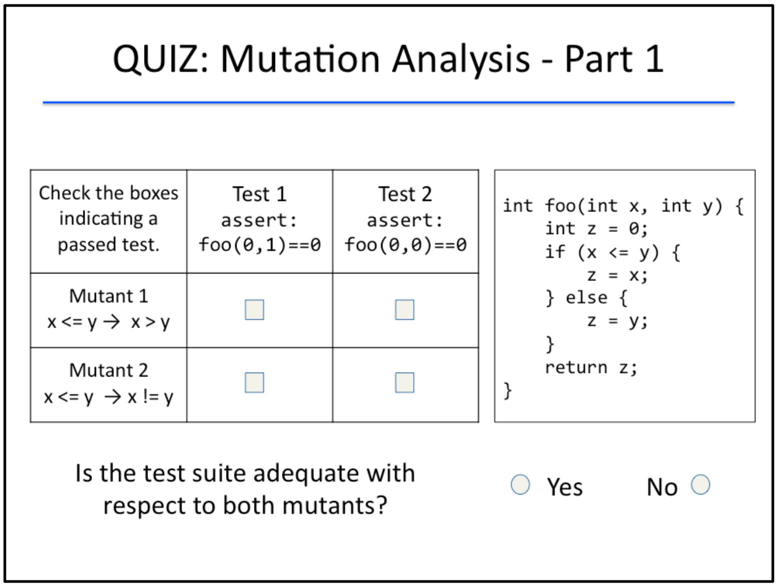
A key advantage of code coverage metrics is their simplicity: they are not difficult to measure. However, they are not perfect. It is possible to attain high code coverage with a test suite yet not discover potential bugs.

A more complex process called mutation analysis can be used to provide more confidence in one’s test suite. Mutation analysis is founded on the “competent programmer assumption,” which is that the program is close to being correct to begin with: the bugs we encounter are likely going to be based on small errors (such as missing a minus sign or replacing a 1 with the letter i) instead of sweeping errors in the program’s overall logic.

The basic idea of mutation analysis, therefore, is to test variations---or mutants---of the program under test. For example, we must switch the direction of a greater-than sign to a less-than sign, or we might add or subtract 1 from some numerical expression.

If our test suite is robust, we should expect each of the mutants to fail some test, while the original program passes all tests. If we discover that certain mutants pass all the tests, it indicates that our test suite is not adequate, and we should therefore add new tests that distinguish the original program from its mutants.

You can read more about mutation testing at the link in the instructor notes.



QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 x <= y → x > y	□	□
Mutant 2 x <= y → x != y	□	□

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite adequate with respect to both mutants? ○ Yes No ○

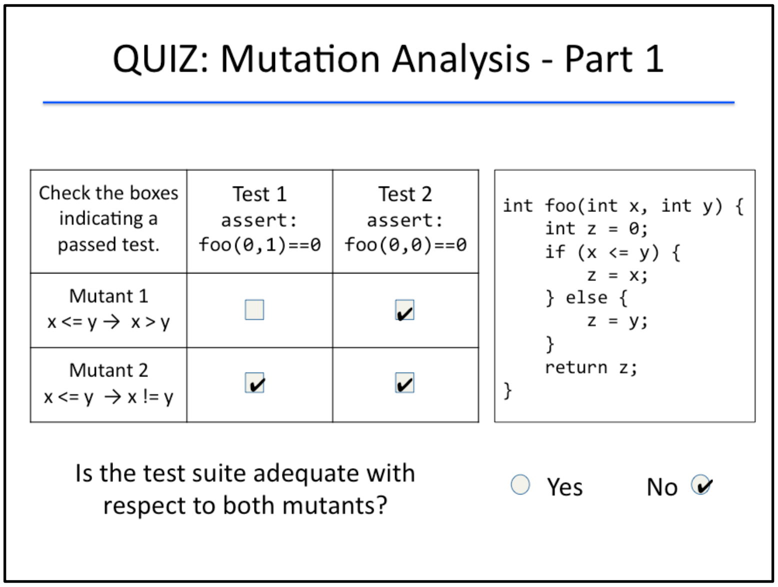
{QUIZ SLIDE}

Here's the function foo() that we saw before, and here are two possible mutations of the function. In the first mutant, the boolean expression x <= y is replaced by x > y. And in the second mutant, x <= y is replaced by x != y.

Let's consider the following test suite comprising these two tests: first, we assert that foo(0,1) == 0; second, we assert that foo(0,0) == 0. (Note that the original function foo will pass both of these tests.)

Check the boxes in the table to indicate which mutants pass which tests.

Then answer the question with either "yes" or "no": is this test suite adequate with respect to these two mutants?



QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 x <= y → x > y	□	☑
Mutant 2 x <= y → x != y	☑	☑

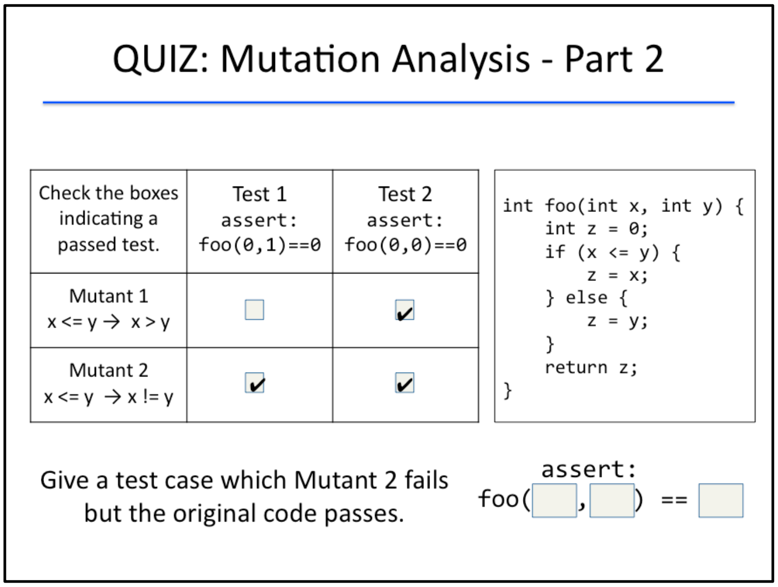
```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite adequate with respect to both mutants? ○ Yes No ☑

{SOLUTION SLIDE}

In the first mutant, where x <= y is changed to x > y, the first test fails because foo(0,1) outputs 1, while the second test passes. So, for this particular mutant, our test suite is robust enough to indicate errors.

However, for the mutant in which x <= y is mutated to x != y, both the tests in our suite pass. This indicates that our test suite is NOT adequate: we need a test case which the second mutant fails but the original code passes.



QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1
	`assert:`
	`foo(0,1)==0`
		Test 2
		`assert:`
		`foo(0,0)==0`

Mutant 1 x <= y → x > y	□	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

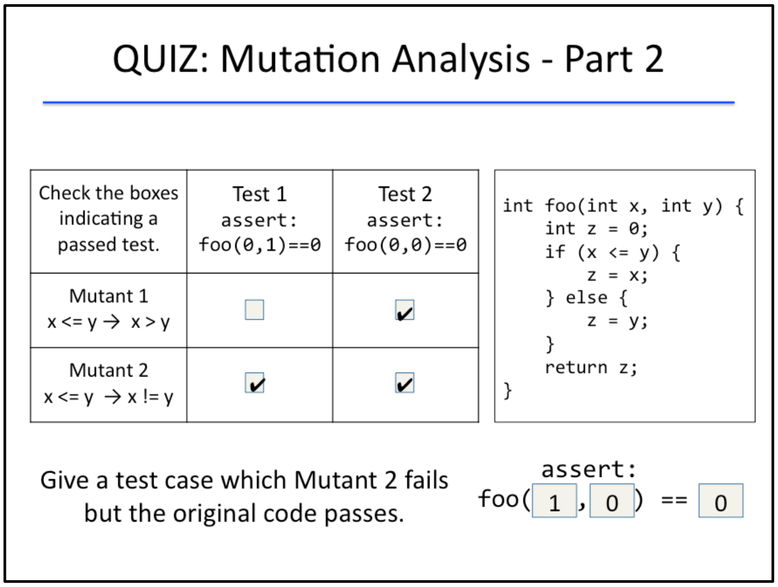
Give a test case which Mutant 2 fails but the original code passes.

```
assert:
foo(□,□) == □
```

{QUIZ SLIDE}

Let's make this test suite more robust with respect to the second mutant. We'll add a statement of the form, assert foo of blank comma blank equals blank.

By filling in the blanks with appropriate numbers, create a test case which Mutant 2 will fail but which the original code will still pass.



QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.	Test 1 assert: foo(0,1)==0	Test 2 assert: foo(0,0)==0
Mutant 1 x <= y → x > y	□	☑
Mutant 2 x <= y → x != y	☑	☑

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Give a test case which Mutant 2 fails but the original code passes.

assert:
foo(1 , 0) == 0

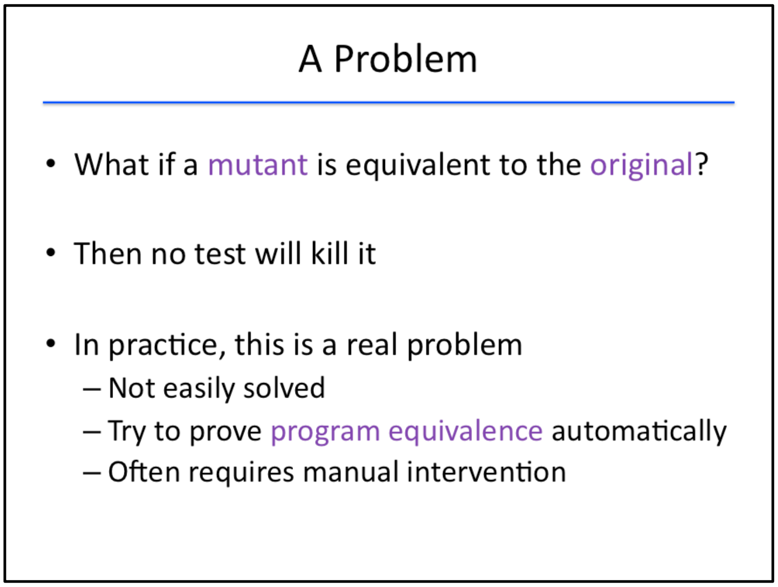
{SOLUTION SLIDE}

Our goal in this quiz is to come up with a choice of x, y, and z so that foo(x,y) == z is true for the original program and false for Mutant 2.

Notice that in the unmodified program, foo returns the minimum of its two arguments. So, in order for foo(x,y) == z to be a true expression, z needs to be the minimum of x and y.

For the mutant program, if x and y are unequal, then foo returns x; if x and y are equal, then foo returns y (which equals x). Thus, foo always returns x. So, in order for foo(x,y) == z to be a false expression, z cannot equal x.

Therefore, we need to choose x, y, and z so that z is the minimum of x and y but not equal to x. This implies we need to choose y and x so that y is less than x. So, any answer where the first input is greater than the second input *and* where the second input is equal to the right-hand side of the expression will work. For example, foo(1,0) == 0.



A Problem

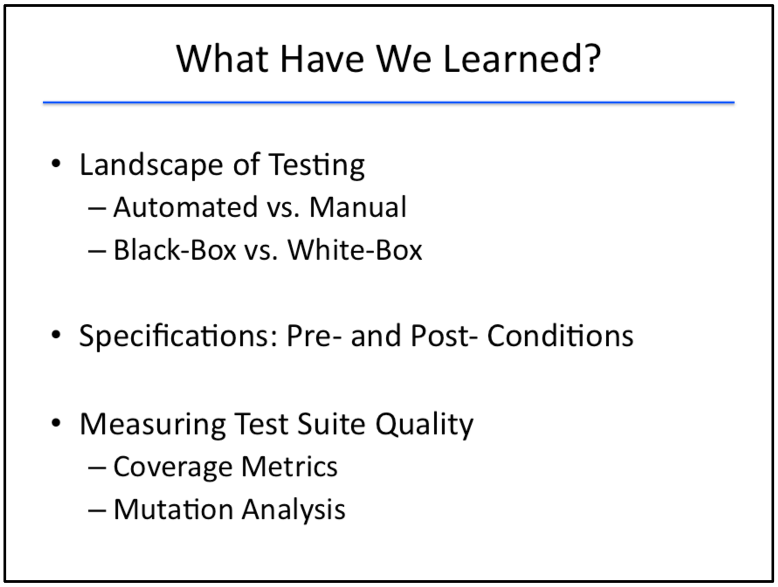
- What if a mutant is equivalent to the original?
- Then no test will kill it
- In practice, this is a real problem
  - Not easily solved
  - Try to prove program equivalence automatically
  - Often requires manual intervention

While mutation analysis is a powerful tool for measuring the quality of a test suite, one problem that could arise is that a mutant is created which is equivalent to the original program. That is, every input to the mutant program will generate the same output as the original program.

Since our testing protocol relies on observing differences between the original program's behavior and mutant programs' behavior, we will find ourselves in a situation where no test kills the equivalent mutant.

If we have such a persistent mutant, it becomes difficult to tell whether it indicates a lack of robustness in our testing or whether it is an equivalent mutant (in which case we can safely ignore the mutant).

This occurs often enough to be a practical problem, and it is not easily solved. We can try to automatically prove that two programs are equivalent, but this problem is undecidable in the general case and often requires a human to intervene and decide whether the mutant in question is indeed equivalent.



What Have We Learned?

- Landscape of Testing
  - Automated vs. Manual
  - Black-Box vs. White-Box
- Specifications: Pre- and Post- Conditions
- Measuring Test Suite Quality
  - Coverage Metrics
  - Mutation Analysis

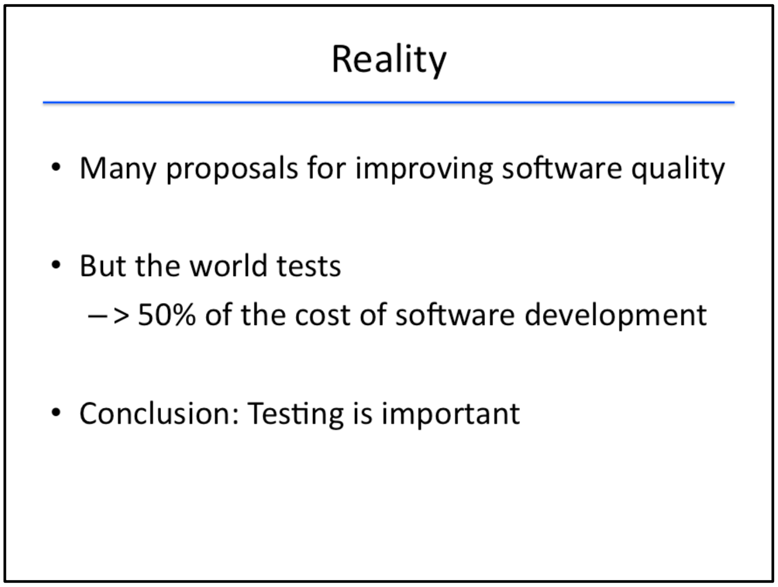
Let’s take a look at what we’ve learned in this introductory lesson to software testing.

We explored the landscape of testing paradigms, focusing specifically on the tradeoffs between automated vs. manual testing and black-box vs. white-box testing. We also noted that most testing protocols will not be strictly at one extreme or the other of these dimensions but rather will lie somewhere in between.

We saw the importance of specifications for software testing, and we learned one way of checking specifications through pre- and post-conditions.

Finally, we discussed two ways to measure the quality of our test suite:

- code coverage metrics, which quantify some facet of how completely our test suite examined all possible situations in which the software might be run,
- and mutation analysis, which attempts to catch bugs introduced by small differences in the code being tested.



Reality

- Many proposals for improving software quality
- But the world tests
  - > 50% of the cost of software development
- Conclusion: Testing is important

There are many proposals for improving software quality, including several that you will learn in this course, many of which attempt to detect program errors before the code even reaches the tester.

However, just as more than half of Microsoft's development costs go to testing, so do more than half the costs of the entire software development industry.

This is because writing error-free programs and verifying programs to be error-free are problems that are inherently undecidable: they can never be fully automated. So there will always be a need to spend resources on testing, and developing tools to improve the efficiency of the testing process.

In the next lesson, we will look at some more techniques to do just that, through the automated generation of test cases.