This is Steve McConnell, author of **Code Complete**.

Measuring isn't enough. That's what testing is: measuring.
To change your results, you have to change your behavior.

(Also referred to today: Michael Feathers, **Working Effectively with Legacy Code**.)

quality

We here a lot about quality and making it. Quality ...

- ... is part of your design
- ... results from your development practices
- ... is monitored and maintained **through the entire software development lifecycle**

You cannot add quality in after the fact.

The more you
invest in quality,
the less time it
takes to make
working software.

Managers and even programmers sometimes think that they “don’t have time” to spec, design, implement or test correctly.

If you don’t have time to do those things, **you will take longer** to produce good systems.

(Quote from Robert L. Glass.)

```
boolean
less_than(PhoneNumber p1, PhoneNumber p2)
{
  ...
}
```

Suppose you have a function for ordering two seven-character phone numbers...

```
boolean
less_than(PhoneNumber p1, PhoneNumber p2)
{
  ...
}
```

10000000 possible values for each arg

Suppose you have a function for ordering two seven-character phone numbers...

```
boolean
less_than(PhoneNumber p1, PhoneNumber p2)
{
  ...
}
```

10000000 possible values for each arg

100000000000000
possible input combinations

Suppose you have a function for ordering two seven-character phone numbers...

```
boolean
less_than(PhoneNumber p1, PhoneNumber p2)
{
  ...
}
```

10000000 possible values for each arg

100000000000000
possible input combinations

at 10000000 tests per second,
exhaustive test takes 155 days

Suppose you have a function for ordering two seven-character phone numbers...

... and that's just one function.

And how do you know

... and how do you know your tests are correct?

Manual tests are manual. They rely on **human** execution.

Automatic tests are **code.** They can be wrong just like our programs!

All our tests can do is to show that there might be an error in the program.

unit test

- exercises one component in isolation condition
- works from the developers perspective, by testing the program's internals

integration test

- exercises the entire system
- works from the user's perspective, by testing the program's overall behavior

regression test

- reruns all tests to ensure that the code still works after a change

Programs that do not have regression tests are difficult — often impossible — to maintain over time.

Suppose we have a "starts with" function:

"eugene" starts with "eug"

"wallingford" does not start with "wall-e"

design a set of unit tests for "starts with"

Specification: startswith returns **true** if the string starts with the given prefix, and **false** otherwise

What if the prefix is the empty string? Yes.

```
Tests = [
    # String  Prefix  Expected
    ['a',     'a',    True],
    ['a',     'b',    False],
    ['abc',   'a',    True],
    ['abc',   'ab',   True],
    ['abc',   'abc',  True],
    ['abc',   'abcd', False],
    ['abc',   '',     True]
]
```

To automate the tests, we can store them in a table of some sort.

Here, the string and prefix make up the **fixture**, the term for the values the a test is run on. The fixture can be as simple as a single value, or as complex as a networked database.

‘Expected’ is the **expected result**.

test cases should be
independent

That is, the outcome of one test should not depend on the outcome of any other. Otherwise, a fault in an early tests can distort the results of later ones.

So, each test should:
- create a new fixture (a new instance)
- perform the operation
- check and record the result

```
passes = 0
failures = 0
for (s, p, expected) in Tests:
    actual = s.startswith(p)
    if actual == expected:
        passes += 1
    else:
        failures += 1
print 'passed', passes, 'out of', \
      passes+failures, 'tests'
```

This is a simple little test harness for my code.

If have to write this many times, I will...

- look for a way to pass in the operation to run on the fixture, so that I can reuse this code
- add code to handle and report errors
- ... and eventually look for a unit test framework that does all this for me

```
> python testStartsWith.py
passed 7 out of 7 tests
```

Having automatic tests makes it so much easier to program confidently.

- confident that your new code works
- confident that your new code has not broken old code
- confident that your changes to old code have not broken old code

The last of these is why unit tests are essential if you want to be able to refactor code.

what if a test case
should cause an error?

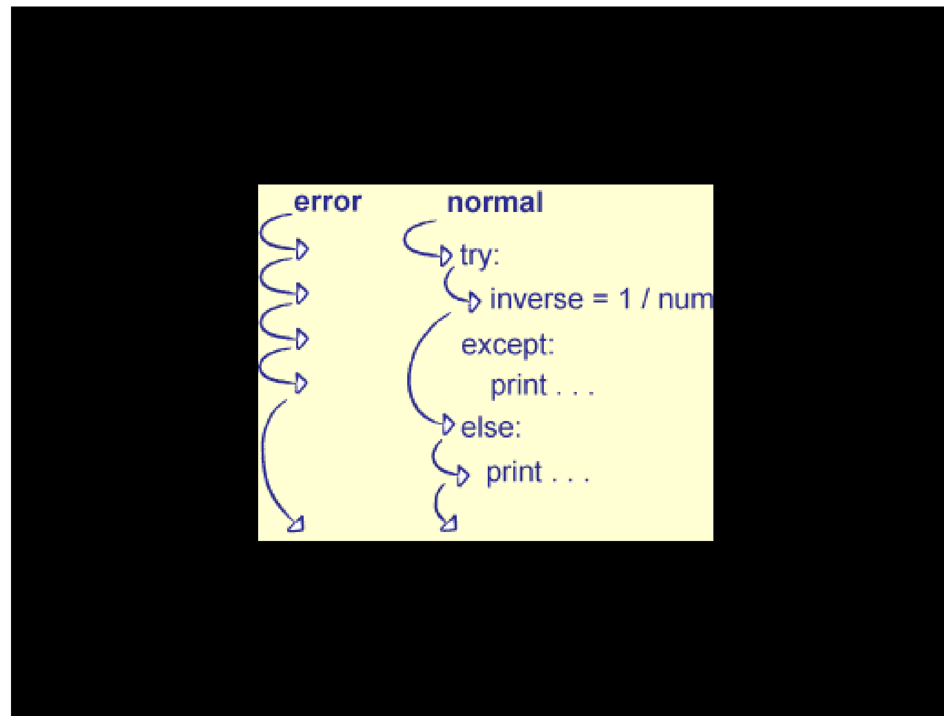
We have to test our programs for their error cases, too!

Use your language's exception handling mechanism, if possible.
Use a testing framework's exception handling mechanism, if possible.

```
for num in [-1, 0, 1]:
    try:
        inverse = 1/num
    except:
        print 'inverting', num, 'caused error'
    else:
        print 'inverse of', num, 'is', inverse
```

Java has a similar mechanism.

```
> python testInversion.py
inverse of -1 is -1
inverting 0 caused error
inverse of 1 is 1
```



Exception handlers work a little bit like **if** statements, with explicit error cases.

Even if you don't program with exception handling ordinarily, you should know its basics so that you can test your code effectively.

But to program defensively, you really **must** program for exceptions!

```
values = [0, 1, 'momentum']
for i in range(4):
    try:
        print 'dividing by value', i
        x = 1.0 / values[i]
        print 'result is', x
    except ZeroDivisionError, e:
        print 'divide by zero:', e
    except IndexError, e:
        print 'index error:', e
    except:
        print 'some other error:', e
```

We can cascade error handling conditions.

In most languages, when a program raises an exception, it creates an object to hold information about what went wrong. This usually contains an error message along with some stack information.

In Python, the programmer can indicate which error to handle by specifying an exception type in the **except** statement. In Java, the programmer **catch**es a specific exception.

```
iMac > python testInversionExceptions.py
dividing by value 0
   divide by zero: float division
dividing by value 1
   result is 1.0
dividing by value 2
   some other error: float division
dividing by value 3
   index error: list index out of range
```

unit tests with errors

We can use exception handling to check for errors in tests: wrap the test code in a **try**/ **excep**t pair.

```
iMac > python testStartsWithExceptions.py
tests: 3
passes: 1
failures: 1
errors: 1
```

Why would we test deliberate errors?

- If the source of input data to a program is unclean, then the program must handle wide-ranging error cases.

- If you are doing **exploratory testing**, then you are trying to learn how a big system works. The tests you write will embody what you learn, including the cases outside the ordinary set of inputs.

(Some people advocate writing tests a la an exploratory tester when learning a new library or a new language.)

tests
are
specs

... or at least **like** specs: “Given these inputs, the program should behave this way.”

test-driven development

So... write the tests first, then the program!?

Sounds backward, but...

This was the #1 request from UNI alumni when asked for testing topics that should be covered.
(alert: the audience was biased, not a full survey)

Why?

a great way to clarify specs

The analyst (or instructor) writes the tests.
The programmer (or student) writes code that passes those tests.

gives the programmer a goal

"Are we there yet?"

The program is done when all the tests run.
This can help programmers avoid "adding just one more feature, then...".

TDD is especially handy when trying to fix faults in existing code, because it forces the programmer to figure out how to re-create the failure.

ensures the tests are written

People are often too tired, or too rushed, to test after coding. The pressure is to move on to the next feature.

But remember: We need to take time to make quality code. TDD is a new habit that helps us engineer quality assurance into the development process, not tack it on at the end.

helps to clarify the interface

... before it is set in stone or in use by the team.

If something is awkward or impossible to test, it can be redesigned before the interface is published.

If something is awkward or impossible to test, **maybe we don't understand the spec** well enough yet!

(... feedback from testing to analysis ...)

reconsider...

```
Tests = [
    # String  Prefix  Expected
    ['a',     'a',    True],
    ['a',     'b',    False],
    ['abc',   'a',    True],
    ['abc',   'ab',   True],
    ['abc',   'abc',  True],
    ['abc',   'abcd', False],
    ['abc',   '',     True]
]
```

A test-driven developer would start with this set of tests as her guide for writing the **starts with** function.

(Yes, Python and Java already have this function built-in!)

In the most extreme form of TDD, found among in extreme programming, the developer would implement her code in steps so small that each version handles **exactly one more** test case!

Sounds crazy, but it works. We'll do a quick demo next time.

TDD is not as new or as extreme as it sounds.

design by contract

Functions carry their specifications around with them.
• Keeping specification and implementation together makes both easier to understand.
• It also improves the odds that programmers will keep them in sync!

A function is defined by:
• its pre-conditions: what must be true in order for the function to work correctly
• its post-conditions: what the function guarantees will be true if its pre-conditions are met
• optional set of invariants: things that are true throughout the execution of the function

This leads to a style of programming called design by contract, which has a long history in the software engineering world.

Pre- and post-conditions constrain how the function can evolve.
We can only...

defensive programming

... is like defensive driving:

"Program as if the rest of the world is out to get you."

The development idea is to "fail early, fail often”.

The smaller the distance between a fault and you detecting it, the easier it will be to find and fix.

We can specify pre- and post-conditions using **assertions**.

An assertion is a statement that something is true at a particular point in a program.

If an assertion's condition is not met, the language raises an exception.

```
def find_range(values):
    '''Find the non-empty range of values \
       in the input sequence.'''
    assert (type(values) is list) \
       and (len(values) > 0)
    left = min(values)
    right = max(values)
    assert (left in values)  \
       and (right in values) \
       and (left <= right)
    return left, right
```

For this function,
- the pre-condition is that its input argument is a non-empty list
- the post-condition is two values from the list such that the first is less than the second

Note that the post-condition is not as demanding as it should be. It does not check that left is less than or equal to all other values in the list, nor that right is greater than or equal to all other values in the list. We are trusting **min** and **max** to be correct.

Besides, the code to check the condition exactly is as likely to contain errors as the function itself. We have to make a trade-off between **time spent testing and guarding** and **the value they produce**!

This is one of the reasons design by contract is not as popular as it might be, and one of the reasons that some programmers object to TDD. But test cases are easier to write than pre- and postconditions, even though writing a complete set is still just as difficult.

don't find bugs.
prevent bugs.

This is real key to quality assurance.