Ghostscript

... was originally written by L. Peter Deutsch in 1986 for the GNU Project. ... The current version of Ghostscript is again available under GPL....

See for more details: http://en.wikipedia.org/wiki/Ghostscript

This was a great piece of software for those of us without access to Adobe's proprietary tools for working with Postscript and PDF.

Ghostscript contained bugs.

Like all software, great or not.

Early versions of the program exhibited bad behavior. Over time, most -- all? -- the bugs were found and fixed.

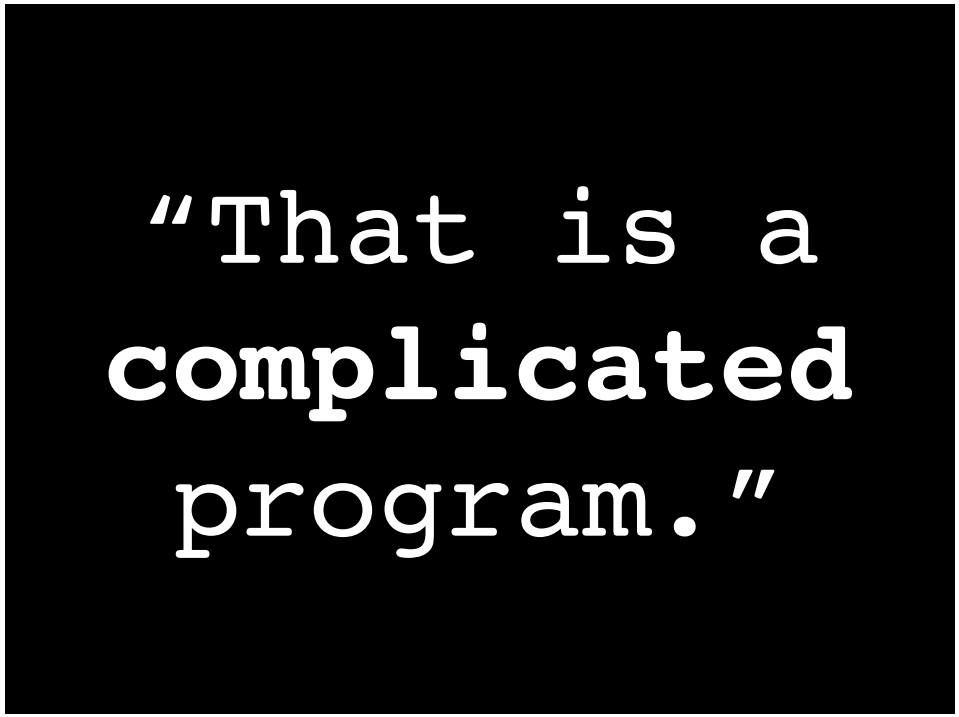
(Then they added more features. Repeat.)

Develop a test plan for `sreadhex()`.

... a function that, in an early version of Ghostscript, contained a fault.

The specification and commentary in the handout come from Brian Marick's book, **The Craft of Software Testing**, Prentice Hall, 1995.

Hints: If you are a Java or Ada programmer, replace uses of pointers with references, access types, or OUT parameters wherever they help your understanding.



Yes, it is. And it is only one function, a utility function at that. Imagine how complicated the whole program is.

Testing matters. The discipline of software engineering recognizes this -- perhaps more than other parts of CS. This is a place where the best part of the engineering metaphor shows through.

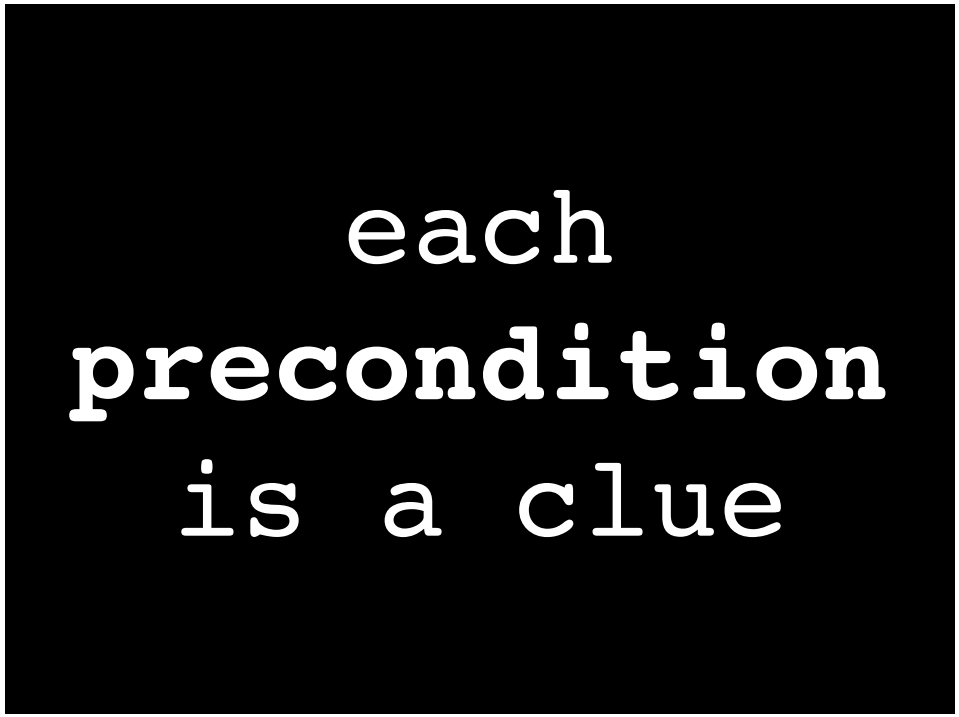
Testing matters so much that the agile approaches move testing near the front of the lifecycle -- maybe all the way to the front, as Extreme Programming does with test-driven development.

It lists explicit pre- and post-conditions. Not all specs are so clear, or so organized around tester considerations.

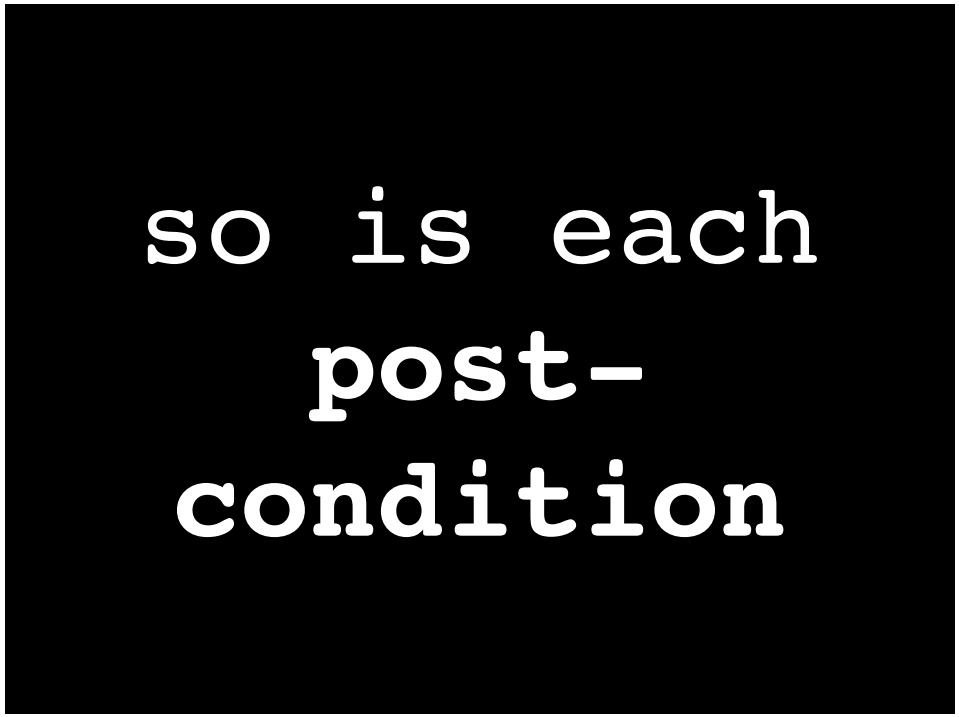
The complexity and foreign feeling of this spec are intentional on my part...

Brian Marick told me, “Some sort of exercise where people have to quickly become expert enough in some non-CS domain would add verisimilitude.” Testers often work in areas they don’t know or understand very well. Knowing that you must learn a new area quickly is important. Knowing that you must is important, too.

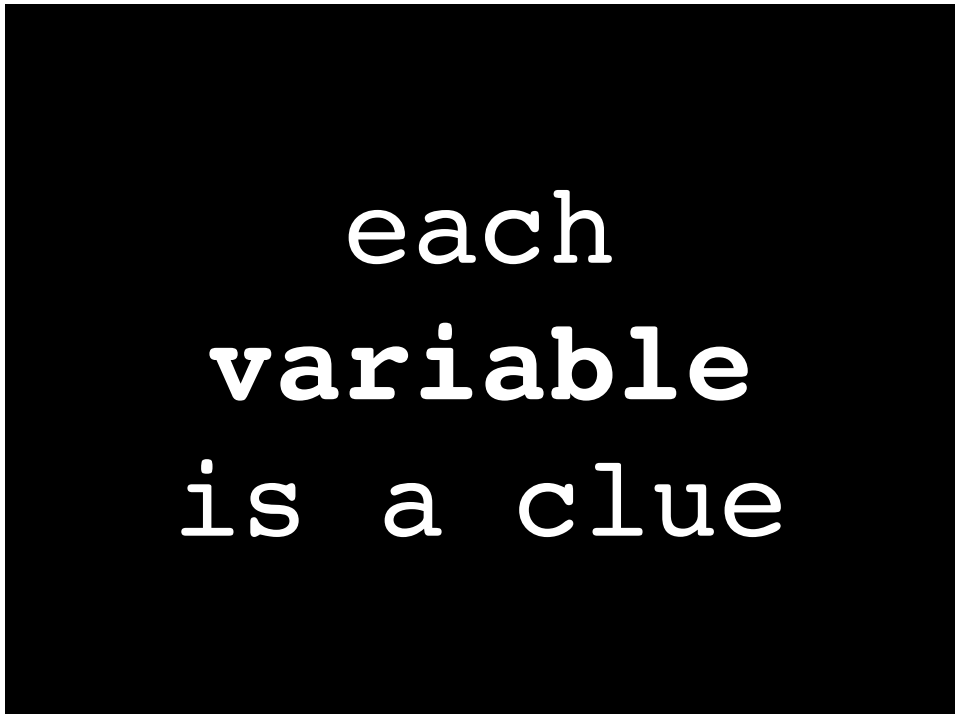
One good thing: Being an outsider can help you be a good tester. You won’t bring as many preconceptions or assumptions to the task. Preconceptions and assumptions can blind us.



A precondition identifies two broad categories of inputs that the program must handle: one where it is met, and one where it is not.



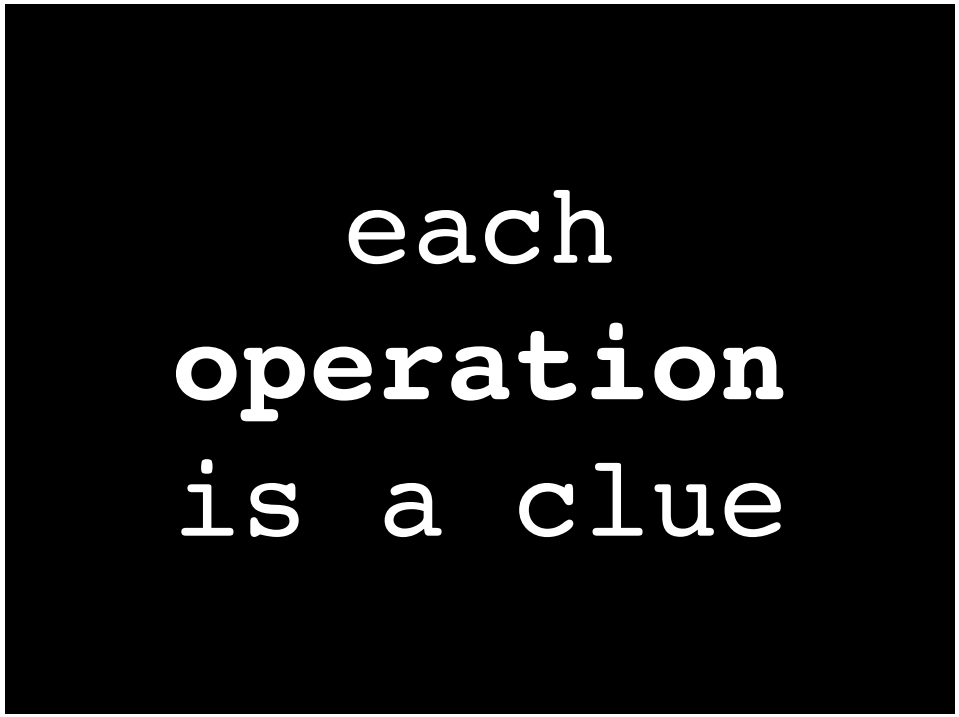
A postcondition identifies the same two broad categories of inputs: one where it is satisfied, and one where it is not.



By variable, we mean any data item that can take different values, including local variables, global variables, arguments, data read from files, and intermediate values mentioned in the spec.

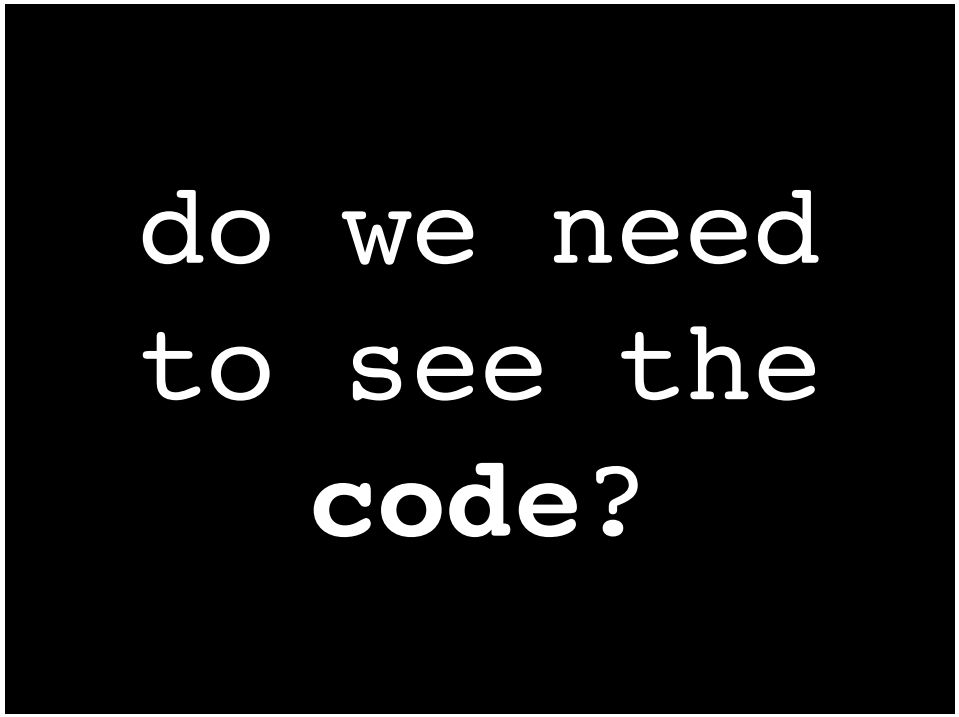
Tests should check for typical misuses of data, such as null or empty strings, or numbers out of range.

How a variable is used matters. If an integer represents a length, then -1 becomes an interesting value -- even though it is not an interesting integer.



Does the function search? sort? process all items in a list? evaluate a boolean condition?

All of these typical operations have their own stereotypical uses and misuses. Our tests should check for them.



The spec is not a complete source of how the program works. Unless we specify modules down to the level of code, there will be a gap between what the spec says and how a program works. **This gap is part of what we must test**!

Code contains implementation details that are irrelevant to the spec but that affect behavior. For example, a sorting routine might use one algorithm for a small datasets and another for large datasets. A good set of tests must exercise both algorithms.

looking at code **too** **early** can be dangerous

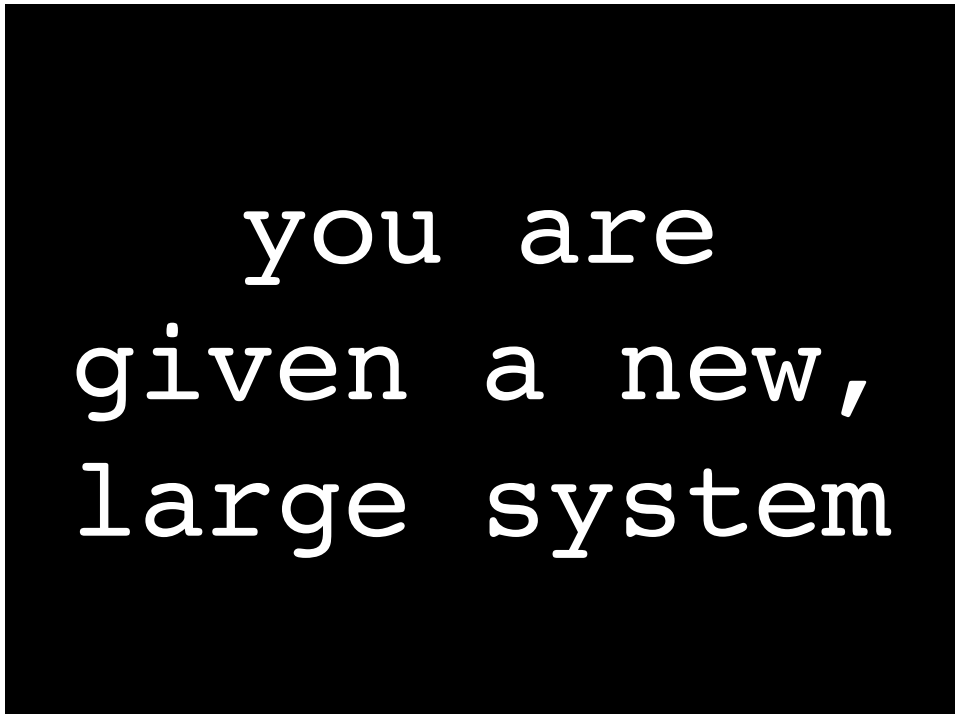
If the programmer misunderstand the spec, then the code will embody the misunderstanding. If you see the code before you fully understand the spec, then the code's assumptions can affect your understanding.

looking at code **too** **late** can be dangerous

If you develop a complete test plan independent of code that already exists, then you will duplicate work done by the programmer to convert the spec into concrete representation. This can make testing more expensive in time and sometimes less effective.

As in designing a program, this involves a trade-off: striking a balance.

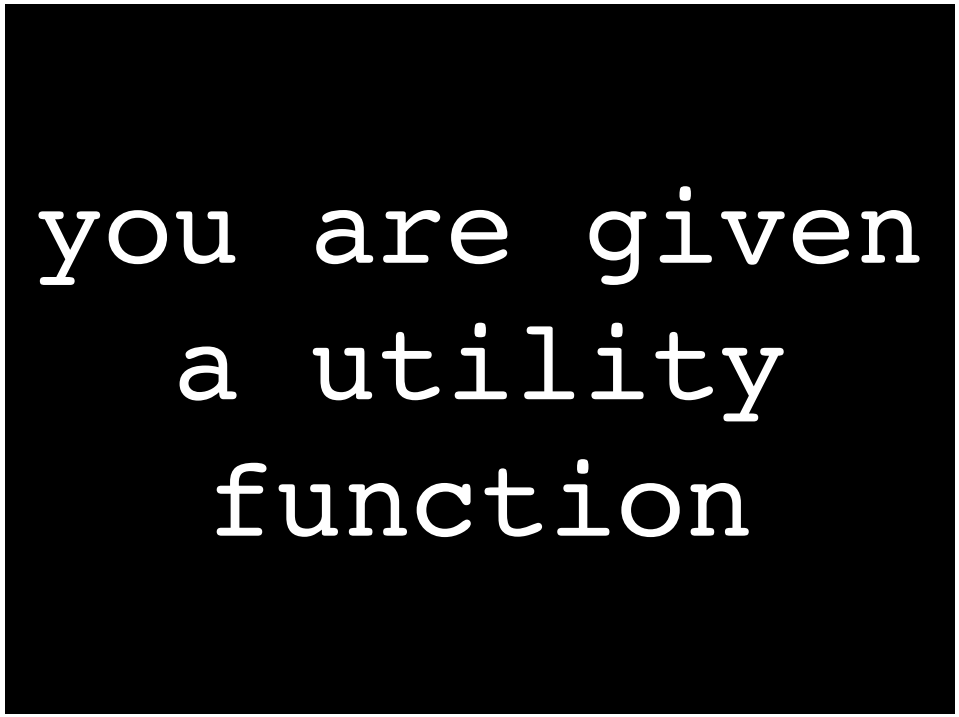
How do you know what the right balance is? “It depends.” Consider these three cases...



In this context, one of the greatest risks bad assumptions that lead to a bad spec and a bad design.

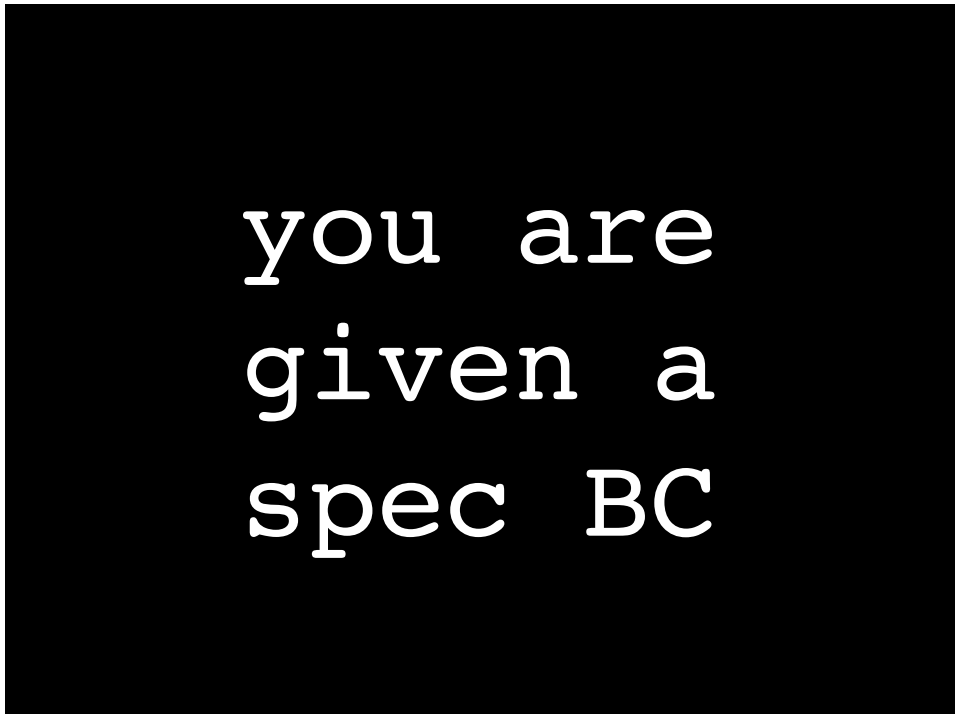
“The user would never want to do **that**!”

Make sure you understand the spec really well before looking at the code. Develop a set of test requirements from the spec, and then augment the test plan with tests from the code.



Bad assumptions are less of a concern. The system is more likely to do something wrong than not to do something. Errors that do something wrong are likely to be typical errors in spec or code (“cliches”).

There is little risk to looking at the code at the same time as the spec, and likely great value. Doing so will save time.



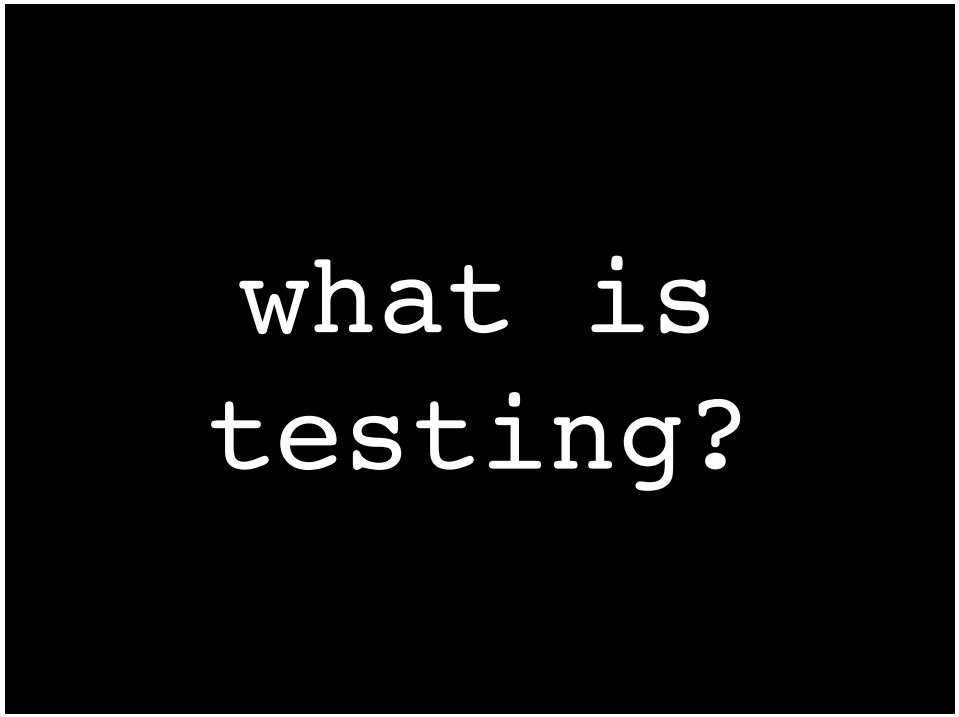
BC == before code

This is the ideal situation for testers!! Too few software organizations operate this way. (This is one of the motivations for the agile approaches: to involve testing much earlier, perhaps even before the code is written.)

Develop a set of test requirements as soon as you have the spec.

- If possible, interact with the programmers. Ask questions such as "What if the user isn't in the permissions file?". Sometimes, this will help the programmers think about things they hadn't thought of yet! This creates a nice synergy between testing and coding, and gives another level of feedback.

- If possible, interact with the analysts, for all the same reasons. This can help to improve the specification itself.



Should I have defined this earlier?

Perhaps, perhaps not. You all have a sense of what it is.

find the bugs in a program

This is what many people think.

But bug is a nebulous term...

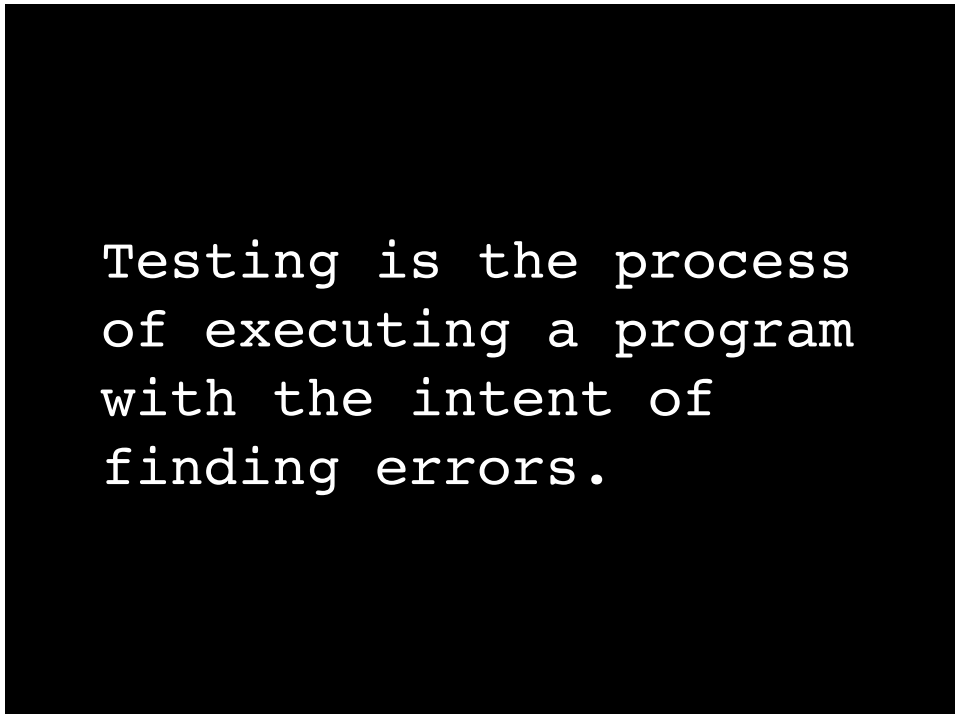
... is a mistake made by a developer: typing, misunderstanding the spec, misunderstanding of what a function does, etc.

... is the difference between a correct program and an incorrect program. A fault usually follows from an error.

Example: The program does not have code checking for an error returned by the `write()` method.

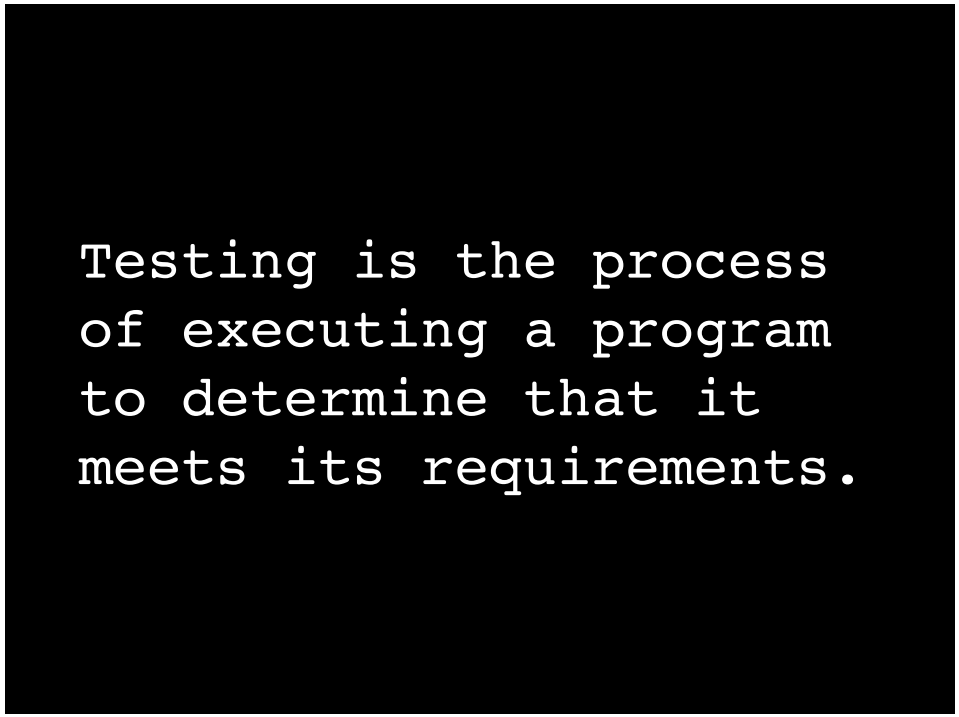
... is the difference between **the results of** a correct program and **the results of** an incorrect program. A fault usually results from an error. A failure usually follows from executing one or more lines of faulty code.

Example: If the `write()` method never signals an error, then the program won't fail as intended.



... much like many students' idea. From The Art of Software Testing, by Glenford Myers.

This is the negative. What is the positive?



... the same idea, stated affirmatively. From The Complete Guide to Software Testing, by Bill Hetzel.

This points out a key distinction:

verification

validation

Does the program do things right?
versus
Does the program do the right things?

Brian Marick: "I failed to do what I intended." versus "I never knew to intend that!"

Considering only the negative side of things -- finding bugs -- tends to miss out on validation, which is often the most common form or fault in a program.

gathering
information
and
comparing them
to expectations

Testing is ...

This is what testers **do**. The other definitions are about **why** they do it.

Gathering information is bigger than “just” identifying input/output pairs to exercise the program. A good tester wants to learn as much about the program, the spec, and the domain as possible. Testers feed this back into the development process wherever possible and use what they learn to produce a more convincing case that the program does or does not meets its specification.

a test is an
experiment
designed to
provide
information
about a program

Again: This is what testers **do**. The other definitions are about **why** they do it.

scientific method in CS

Is Computer "Science"?

What kind of science is "computer science"?

- algorithms -- mathematics
- operating systems?
- programming languages?
- software engineering?

Software engineering studys how we make software -- software process -- rather than the computational processes that define the field more generally.

We should make software engineering more than...

If you can only read one book, this is probably the single best summary of empirical SE