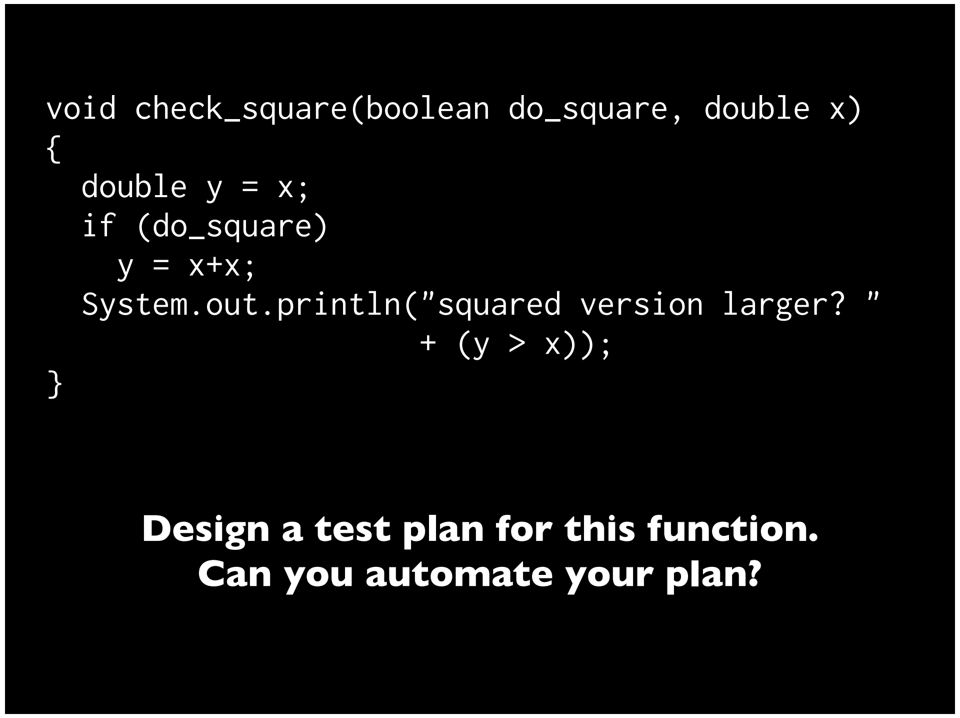
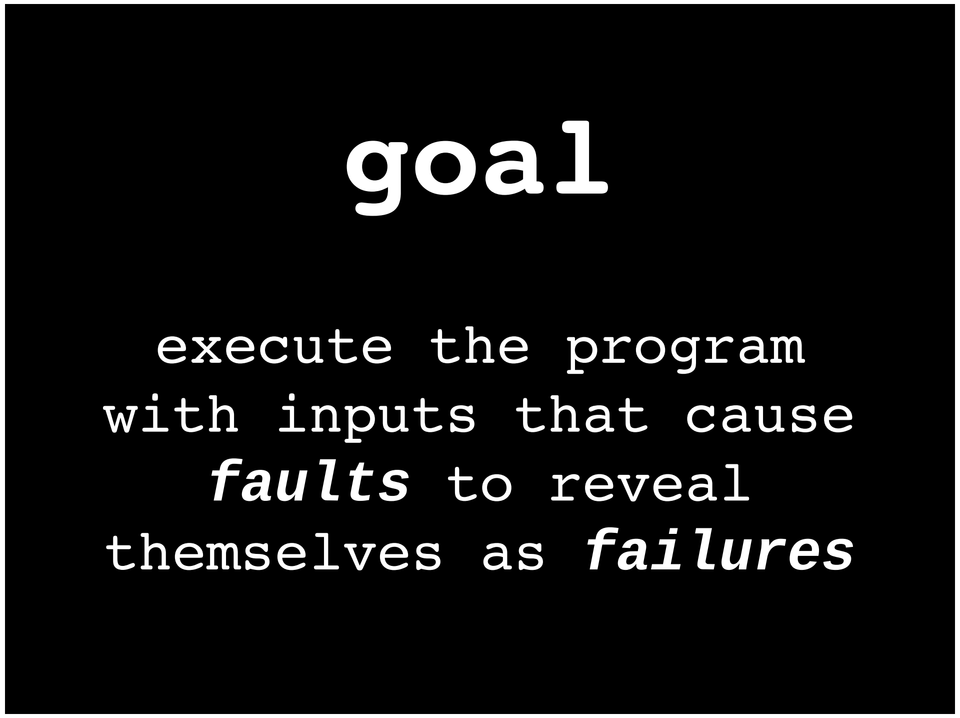


```
void check_square(boolean do_square, double x)
{
  double y = x;
  if (do_square)
    y = x+x;
  System.out.println("squared version larger? "
                     + (y > x));
}
```

Design a test plan for this function.
Can you automate your plan?

Bonus points available if your plan uncovers a flaw!
(Run student tests against the code...)

The examples and many of the concepts in this session again come from Brian Marick's book, **The Craft of Software Testing**, Prentice Hall, 1995.



goal

execute the program with inputs that cause ***faults*** to reveal themselves as ***failures***

What is the fault in `check_square()`?

A `+` was used where a `*` should have been used.

What are the **ideal fault conditions**?

reachability

The input must cause the faulty statement to be executed.
Such inputs satisfy the **reachability condition**.

In `check_square()`, `do_square` must be true.

If there are two faults, is it always possible to isolate them with separate test cases?
If we cannot, how can we still debug the code effectively?

necessity

The input must cause the faulty statement to produce a different result than the correct statement. This will cause the program to enter into an incorrect internal state.

Such inputs satisfy the **necessity condition**.

In `check_square()`, `x = 0` will cause `y` to have   the correct value: `0 == 0`.   !
In `check_square()`, `x = 1` will cause `y` to have an incorrect value: `2 != 1`.
In `check_square()`, `x = 2` will cause `y` to have   the correct value: `4 == 4`.   !

The program can compute the correct answer in an incorrect way.

The necessity condition for `check_square()` is `(x != 0 && x != 2)` .

propagation

The input must cause the incorrect internal state to propagate to an observable failure.

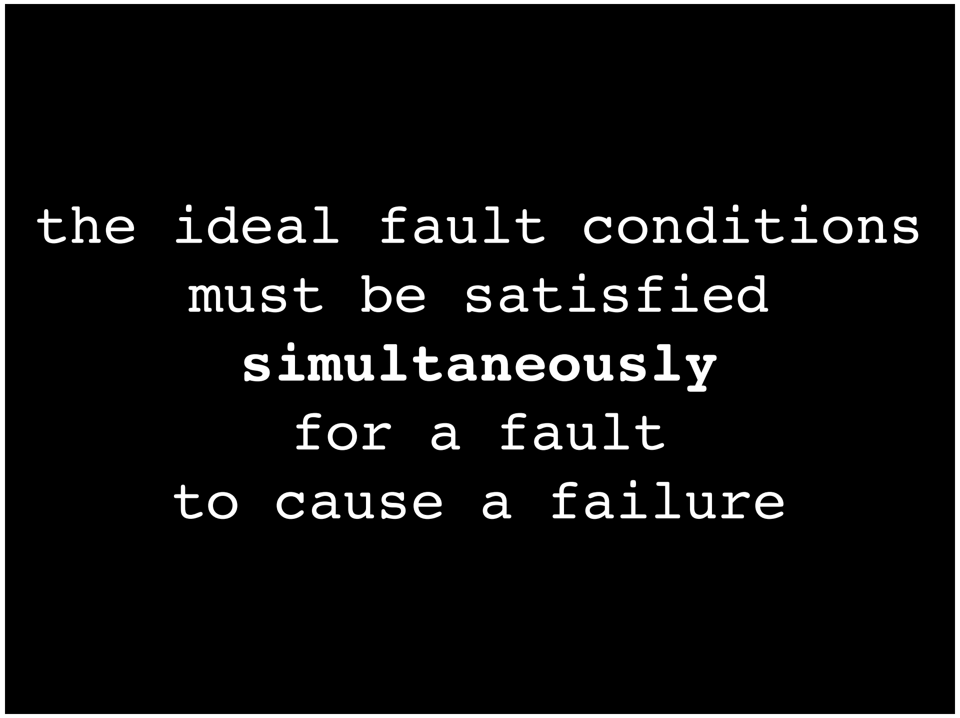
Such inputs satisfy the **propagation condition**.

In `check_square()`, `x = 3` will cause `y` to have an incorrect value: `6 != 9`.
However, both 6 and 9 are > 3, so the program will print

```
    squared version larger? true
```

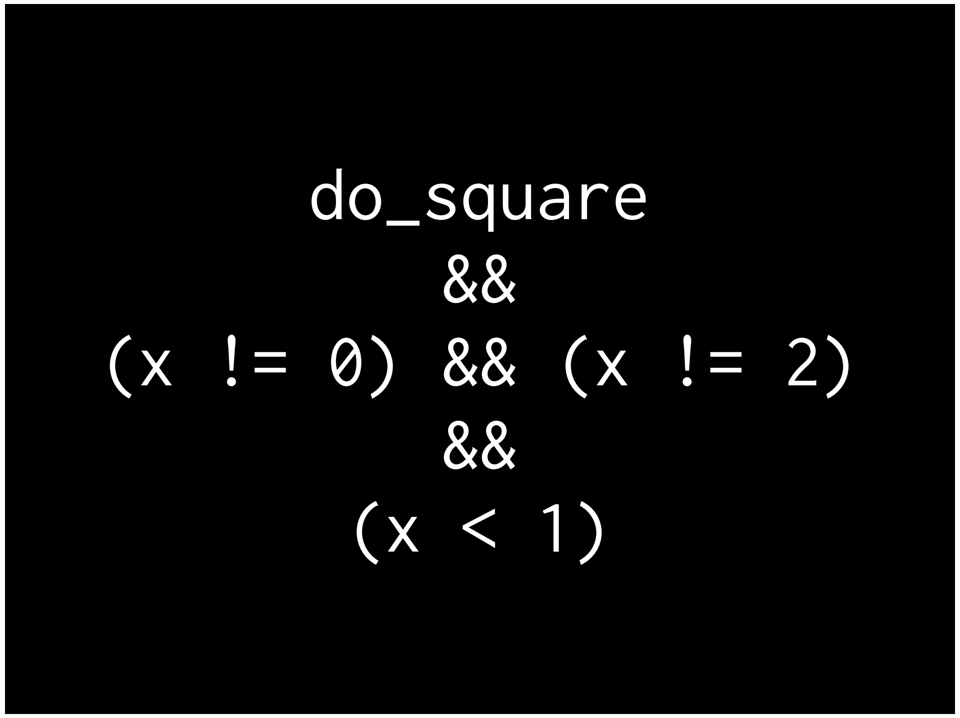
There is no failure. The faulty program produces the same output as a correct one. So 3 does not satisfy the propagation condition.

The program's observable behavior may be the correct even when it is in an incorrect internal state.



the ideal fault conditions
must be satisfied
simultaneously
for a fault
to cause a failure

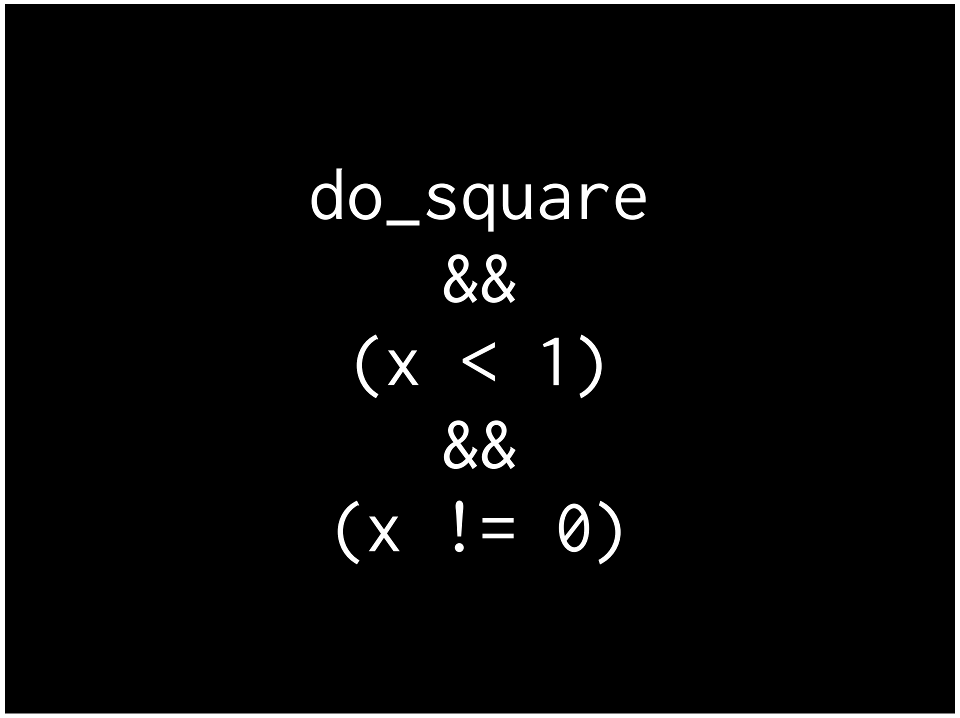
so…



```
do_square
&&
(x != 0) && (x != 2)
&&
(x < 1)
```

We conjoin the reachability, necessity, and propagation conditions to find the ideal fault conditions for this particular bug.

From, there we can simplify...



```
do_square
&&
(x < 1)
&&
(x != 0)
```

This gives us the simplest conditions for finding this particular bug.

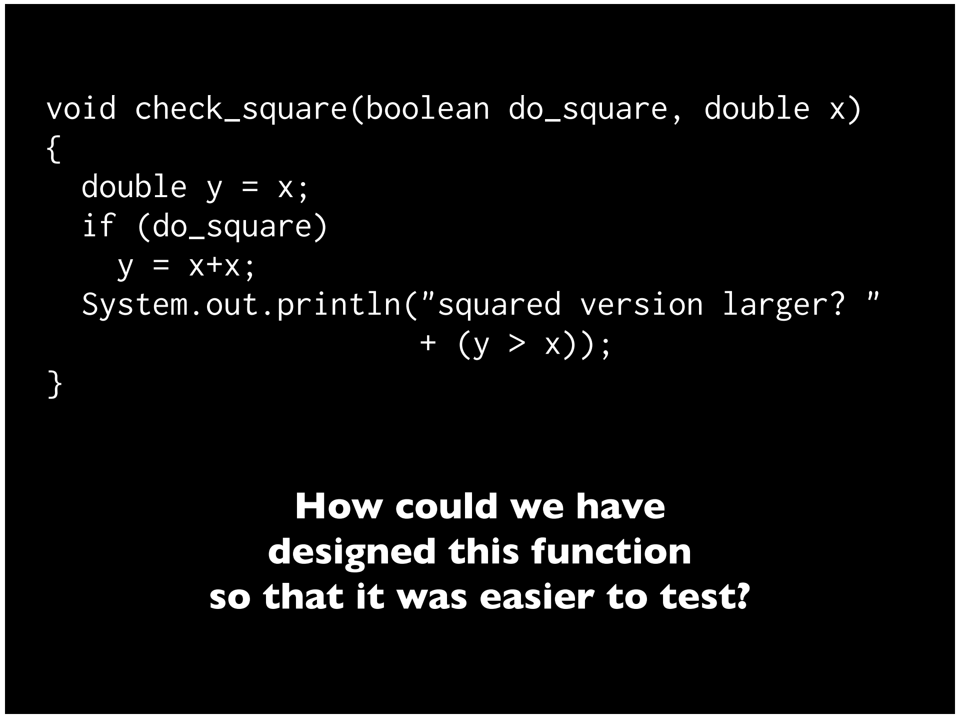
What is that phrase I keep saying?

this particular bug

The ideal fault conditions are **fault-specific**. They follow from the errors in the program.

Unfortunately, we do not always, or even usually, know what or where the faults are.

Our goal as testers is to approximate the ideal: to create test cases that are likely to find errors and faults and to expose them as failures.



```
void check_square(boolean do_square, double x)
{
  double y = x;
  if (do_square)
    y = x+x;
  System.out.println("squared version larger? "
                     + (y > x));
}
```

How could we have designed this function so that it was easier to test?

We could separate the computation from the I/O behavior:

```
void compute_square(boolean do_square, double x)
{
  double y = x;
  if (do_square)     // could leave this test
    y = x+x;         // in check_square
  return y;
}

void check_square(boolean do_square, double x)
{
  double y = compute_square(do_square, x);
  System.out.println("squared version larger? "
                     + (y > x));
}
```

What is the trade-off on where we do the check for do_square?

- This solution isolates I/O behavior in check_square().
- The alternative isolates the domain decisions in check_square().

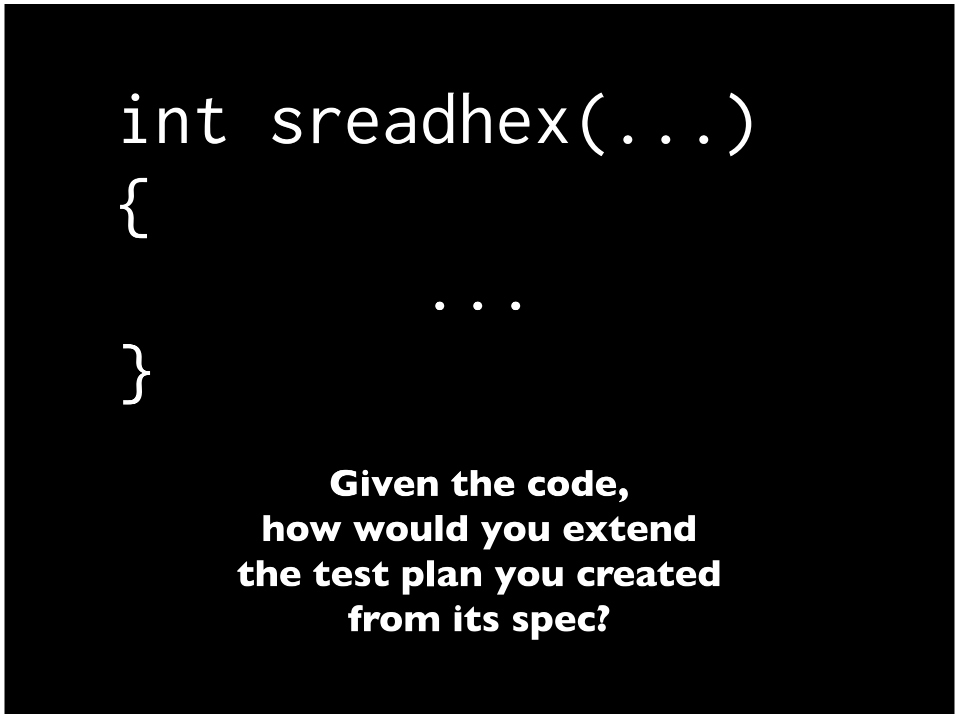
What is the trade-off on one function versus two?

- One function is faster at run-time.
- Two functions are easier to test, and maybe to understand.

As much as possible, let your tools work for you.

If your language and compiler support in-lining of functions, tease apart your functions into as many pieces as possible, for testing and understandability. Let the **compiler** put them back together with in-lining.

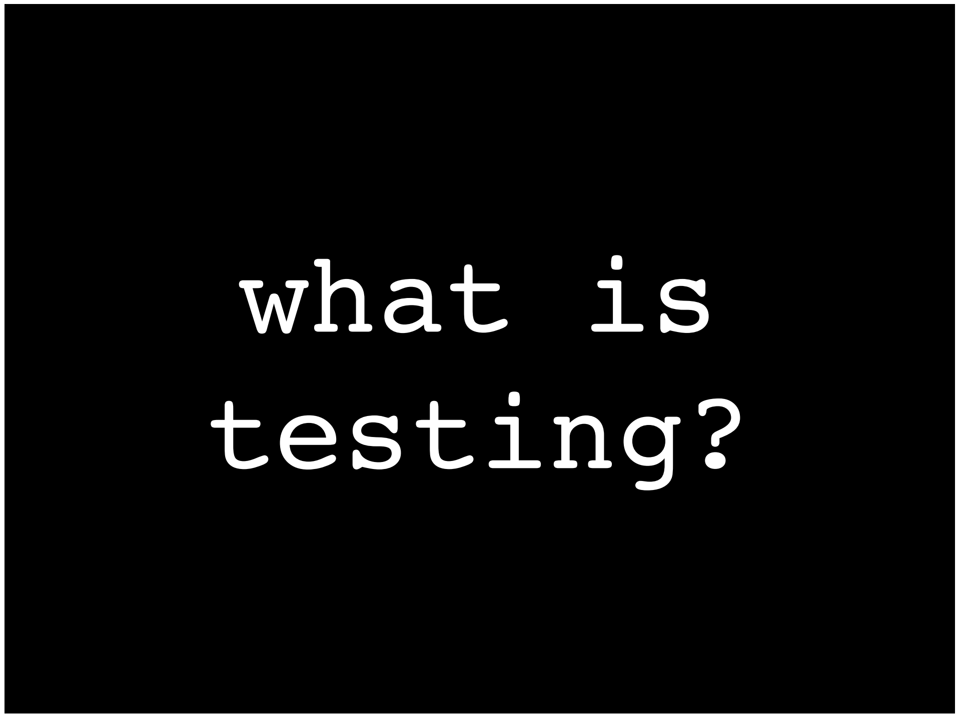
When plausible, assume that run-time speed is less important than testing and



```
int sreadhex(...)
{
            ...
}
```

**Given the code,
how would you extend
the test plan you created
from its spec?**

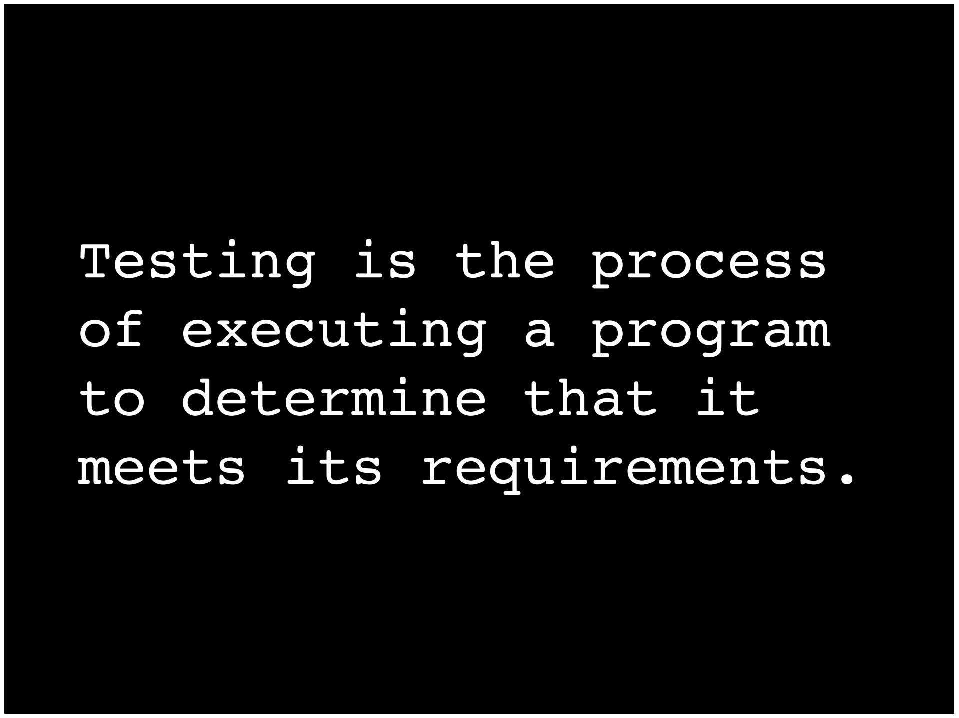
In what ways does having the code give us better insight into testing this function?



what is testing?

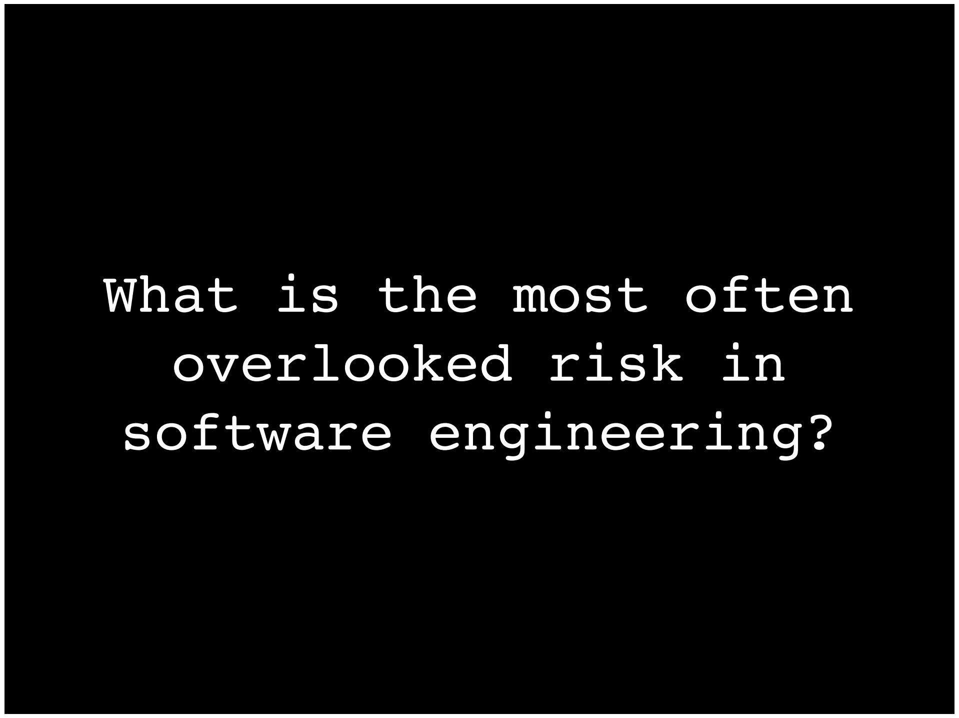
Should I have defined this earlier?

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


Testing is the process of executing a program to determine that it meets its requirements.

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

This points out a key distinction:

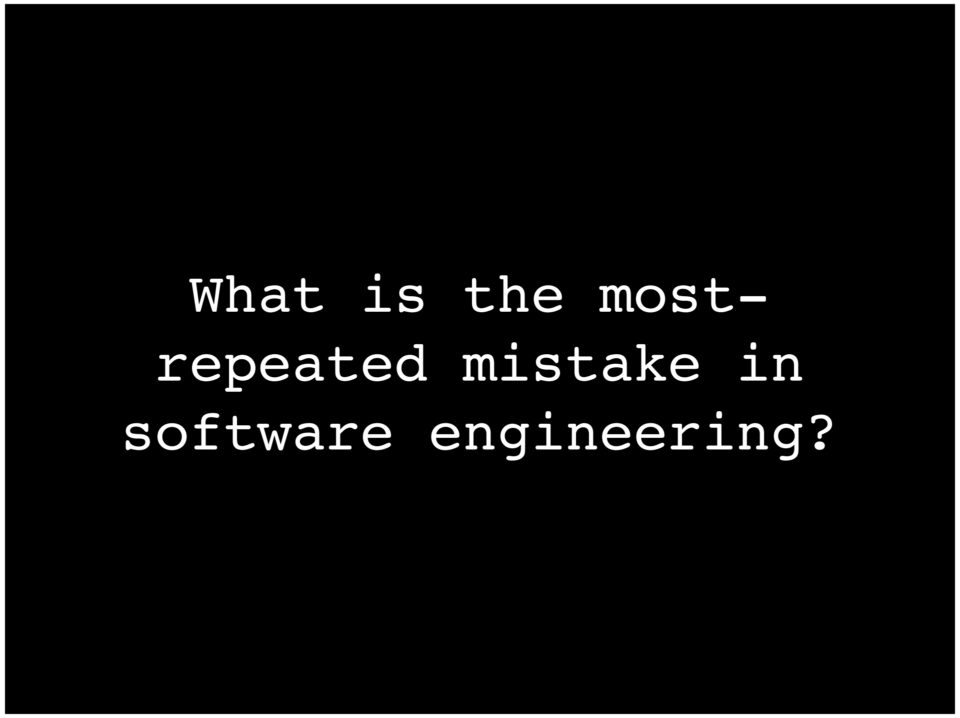

What is the most often overlooked risk in software engineering?

From an interview with software engineering pioneer David Lorge Parnas.
http://www.sigsoft.org/SEN/parnas.html

[FILL IN DETAILS ON PARNAS]

incompetent programmers

“There are estimates that the number of programmers needed in the U.S. exceeds 200,000. This is entirely misleading. It is not a quantity problem; we have a quality problem. One bad programmer can easily create two new jobs a year. Hiring more bad programmers will just increase our perceived need for them. If we had more good programmers, and could easily identify them, we would need fewer, not more.”

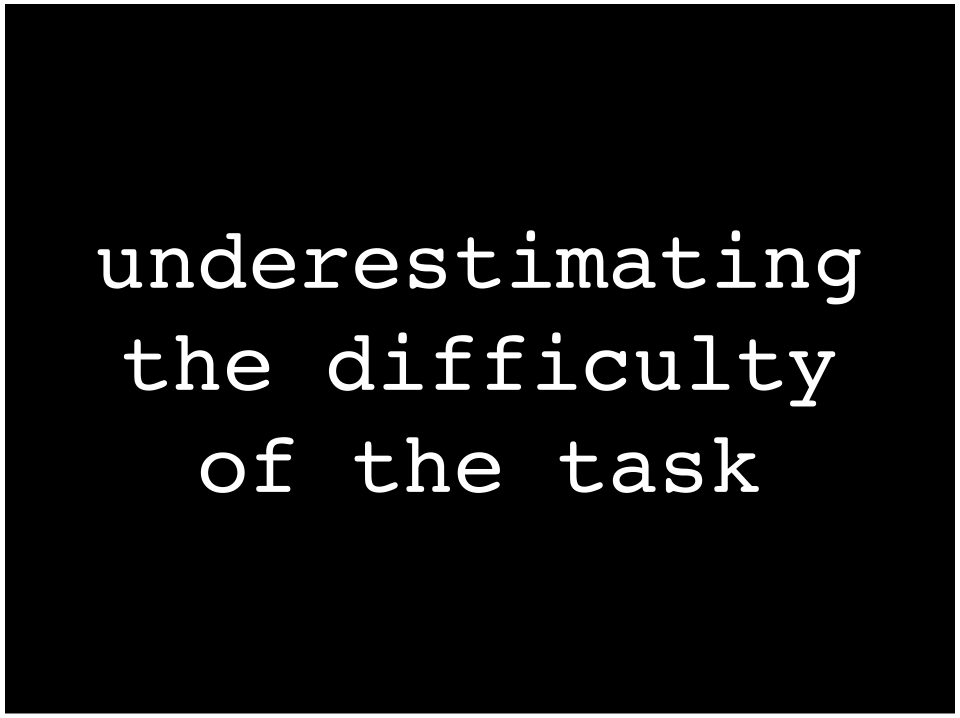


What is the most-repeated mistake in software engineering?

From the same interview with Parnas.
http://www.sigsoft.org/SEN/parnas.html

over-confidence

“People tend to underestimate the difficulty of the task. Overconfidence explains most of the poor software that I see. Doing it right is hard work. Shortcuts lead you in the wrong direction and they often lead to disaster.”



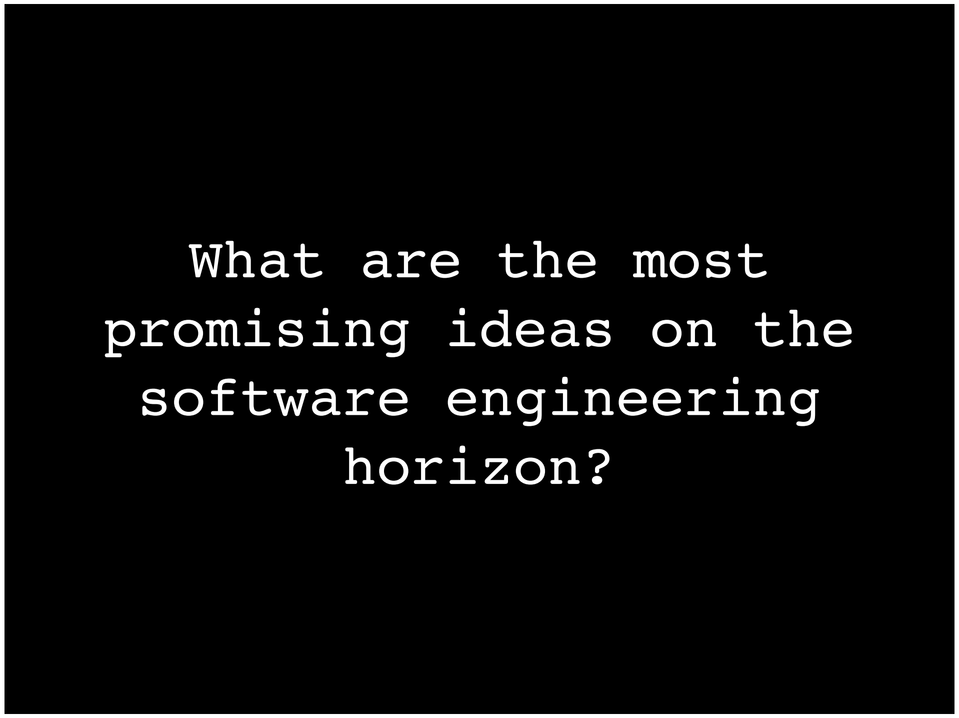
underestimating the difficulty of the task

This is a repeated theme in software engineering and programming.

Hubris, good and bad.

Hard work.

Testing is a great example.



What are the most promising ideas on the software engineering horizon?

From the same interview.

they are already here

“I don't think that the most promising ideas are on the horizon. They are already here and have been here for years but are not being used properly.”