Random input testing with R

Patrick Burns
http://www.burns-stat.com

2011 August

Given at useR!2011 at the University of Warwick on 2011 August 17 in the Programming session, Uwe Ligges presiding.

The goal is for us to have fewer bugs in our code.

Photo from istockphoto.com

We'll start with a bit about the application that drove me in the direction that I've come.

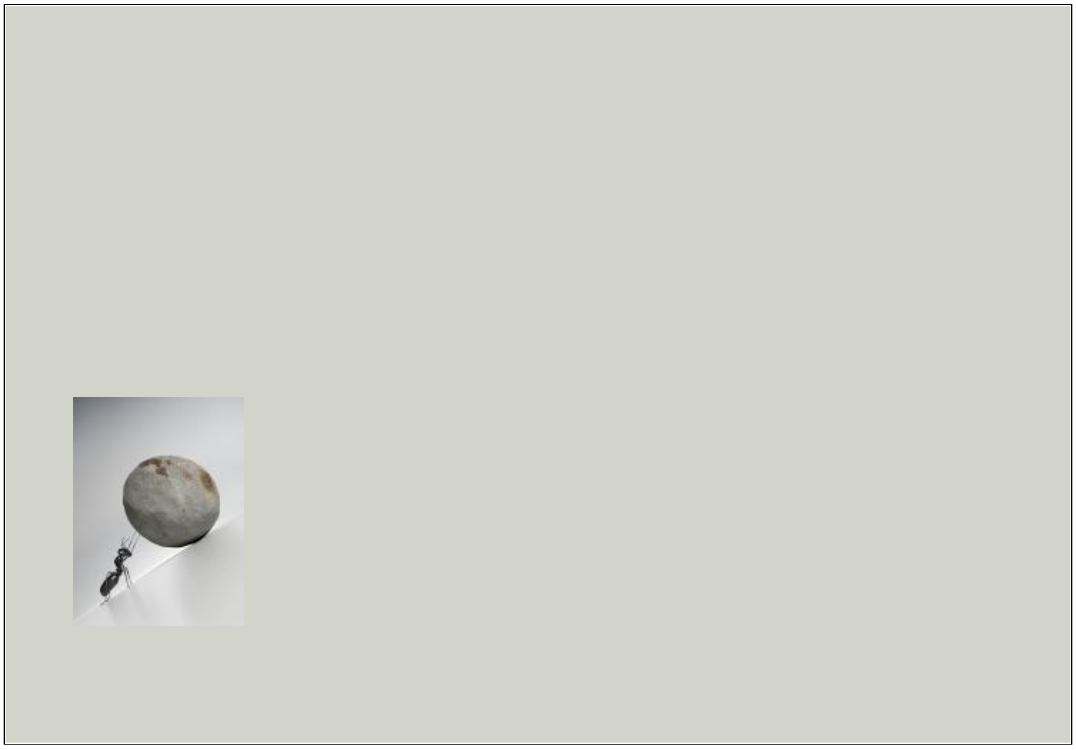
The application is portfolio optimization in finance. The textbook version of this is quadratic programming – that's an easy problem.

The reality is that it is a species of the knapsack problem: what is the best bundle of stuff that you can carry with you? That's a hard problem.

How hard?

Let's think about doing it via brute force.

Photo by David Weekly via everystockphoto.com



We’ll take a small problem.

Photo via istockphoto.com

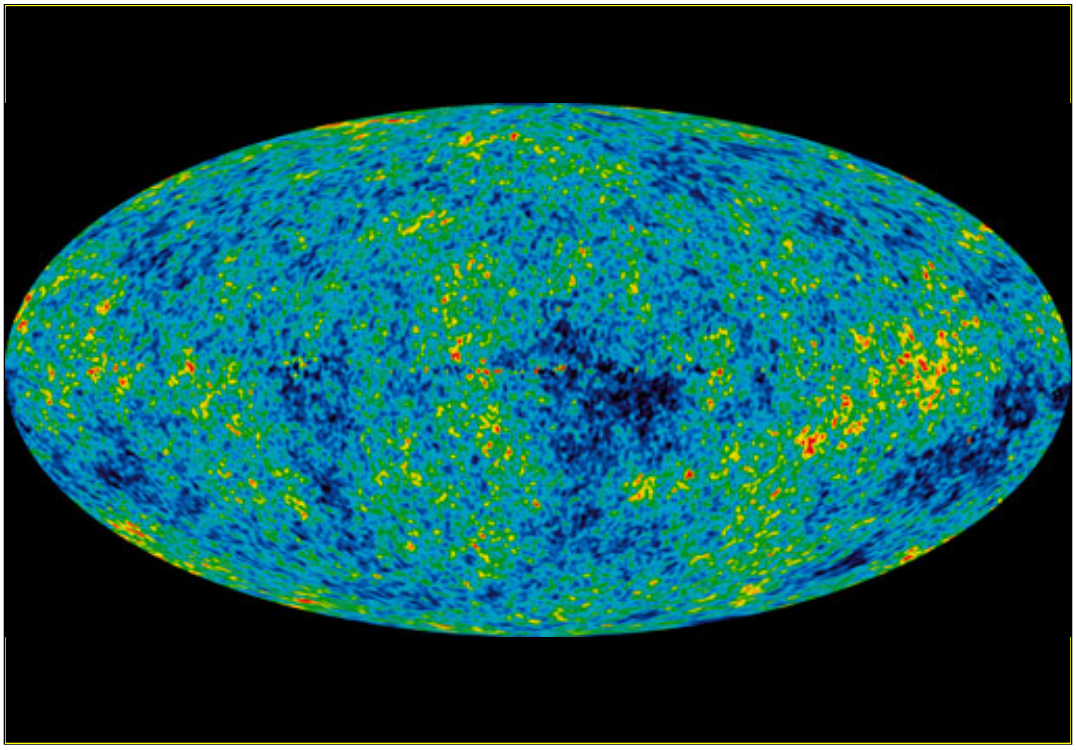
We’ll make some blue sky, absolutely optimistic assumptions.

Photo via istockphoto.com

Unit of time

Now we need a unit of time to state how long this process will take.

I have one in mind, what is it?



The first guess from the audience was right: the age of the universe.

There is a combinatorial explosion that makes this a fantastically hard problem.

Wilkinson Microwave Anisotropy Probe image from NASA

Not everyone is keen on the wait.

Photo from istockphoto.com

So in my code I have a number of shortcuts to speed up the process.

Nothing could possible go wrong here.

Photo by jurvetson via everystockphoto.com (I think)

Utilities

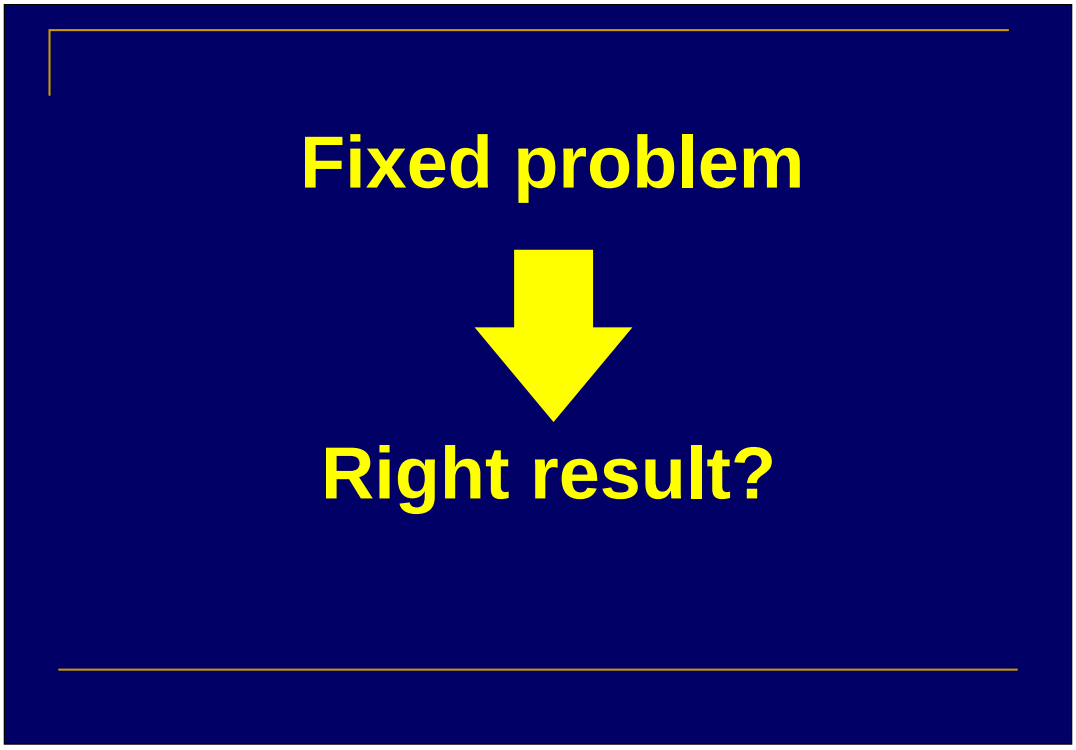
- Maximise information ratio
- Minimum variance
- Maximum return
- Mean-variance utility
- Mean-volatility utility
- Minimize distance to a portfolio
- With or without transaction costs
  - Linear costs
  - Polynomial costs
  - Multiple terms with arbitrary exponents
  - Separate costs for long-buy, long-sell, short-buy, short-sell
- With or without multiple expected return vectors
- With or without multiple variance matrices
- With or without multiple target portfolios
- Choice of quantile in multiple utility cases
- Can use weights in multiple utility cases

When doing optimization, we need a utility. So pick one.

There will also be a set of constraints which is a subset of all possible constraints.

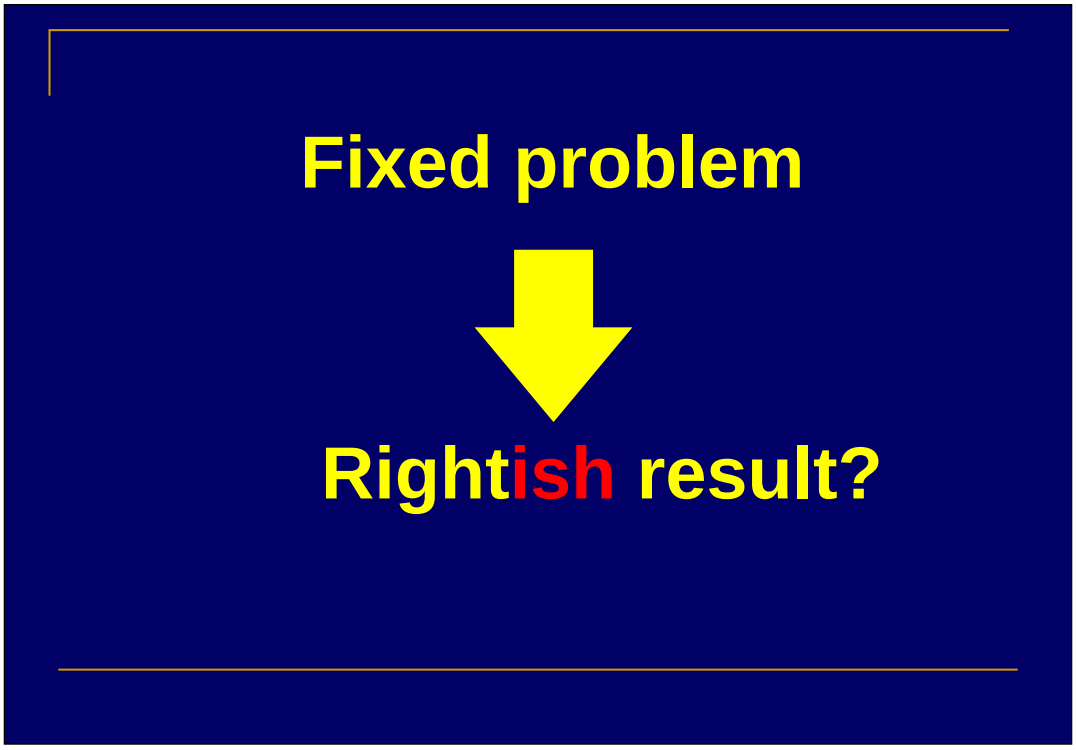
So there is a combinatorial explosion of inputs that is similar to the combinatorial explosion that makes it a hard problem in the first place.

And we want to test to see if the code works.



The standard approach to software testing is to produce a fixed problem and ask if we get the correct answer.

Perhaps you can spot that already I have a problem.



If it takes billions of years to get one right answer, perhaps my test suite will be a bit thin for a while.

I can’t ask if I have the right answer. I can only ask if the answer is sort of right.

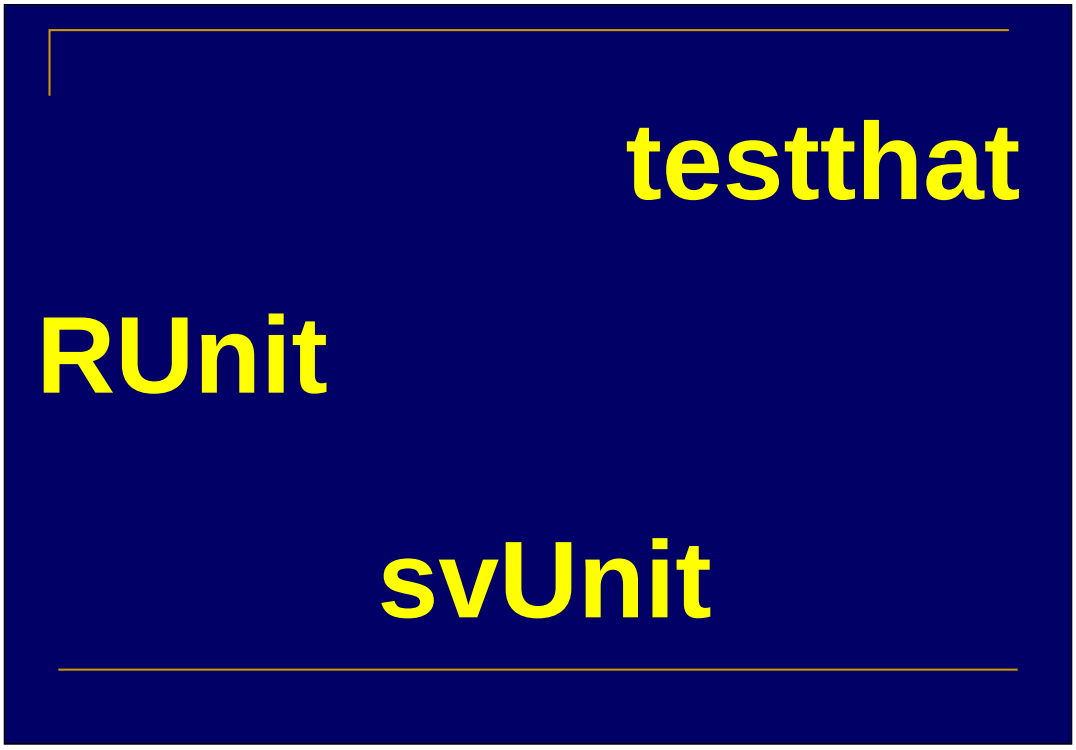
Rightish answers

- Sanity check shortcuts
- Sanity check result
- Close to best observed?

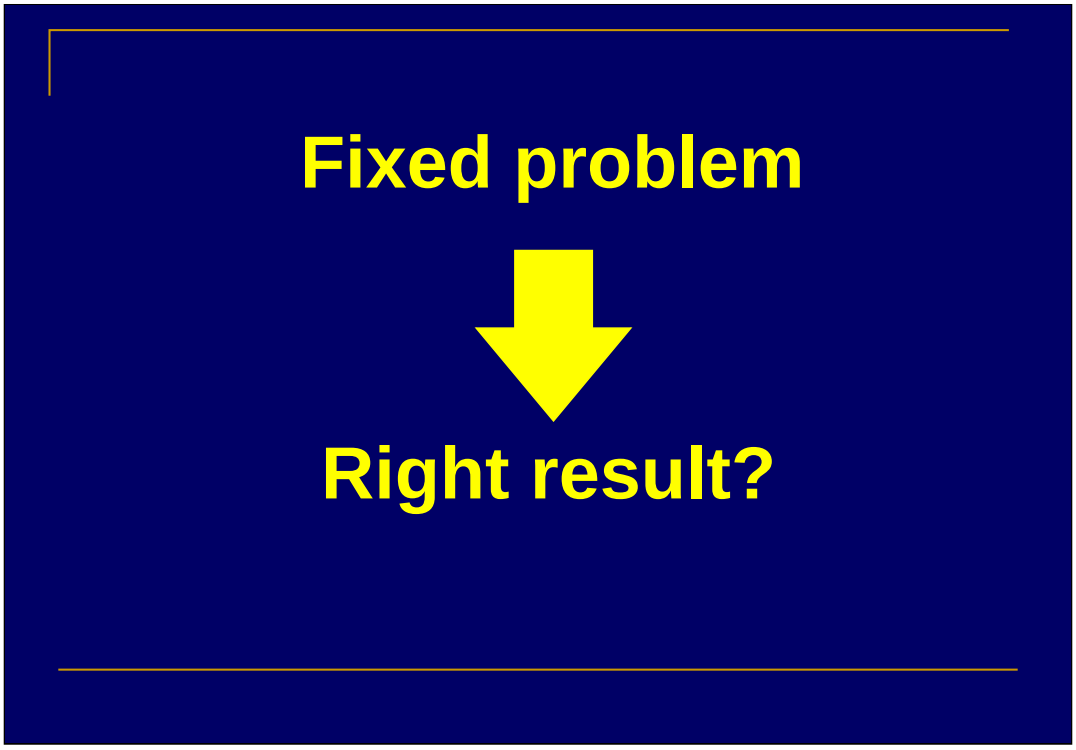
To see if an answer appears right, there is a large number of checks in test mode that squawk if something doesn't look right. In particular almost all shortcuts are also computed the long way in test mode and the two results compared.

There are checks to see if the result makes sense.

Since my optimization algorithm is random, it will potentially get different answers when using different seeds. Hence we can see if the current answer is close to the best answer that has been observed. This gives us a sense of how well the algorithm works on the particular problem.



If you are creating a standard type test suite for R code, then there are a few packages that can help you.



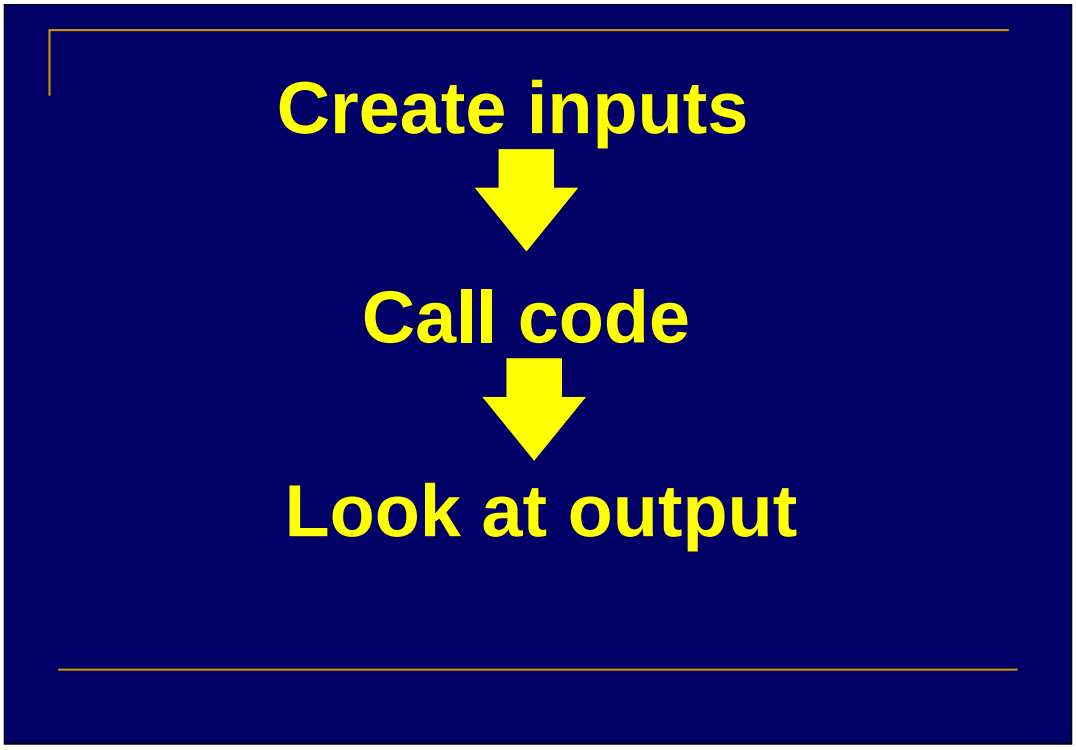
I have a standard test suite. But no matter how expansive my suite becomes, I will only ever test a tiny fraction of the possible combinations of inputs.

And writing a standard test suite is quite labor-intensive.

Plus there can be bias in the test. As someone said to me after the talk, we may have an unconscious desire NOT to find bugs.

Hence looking for alternatives to standard testing seems like a worthwhile endeavor.

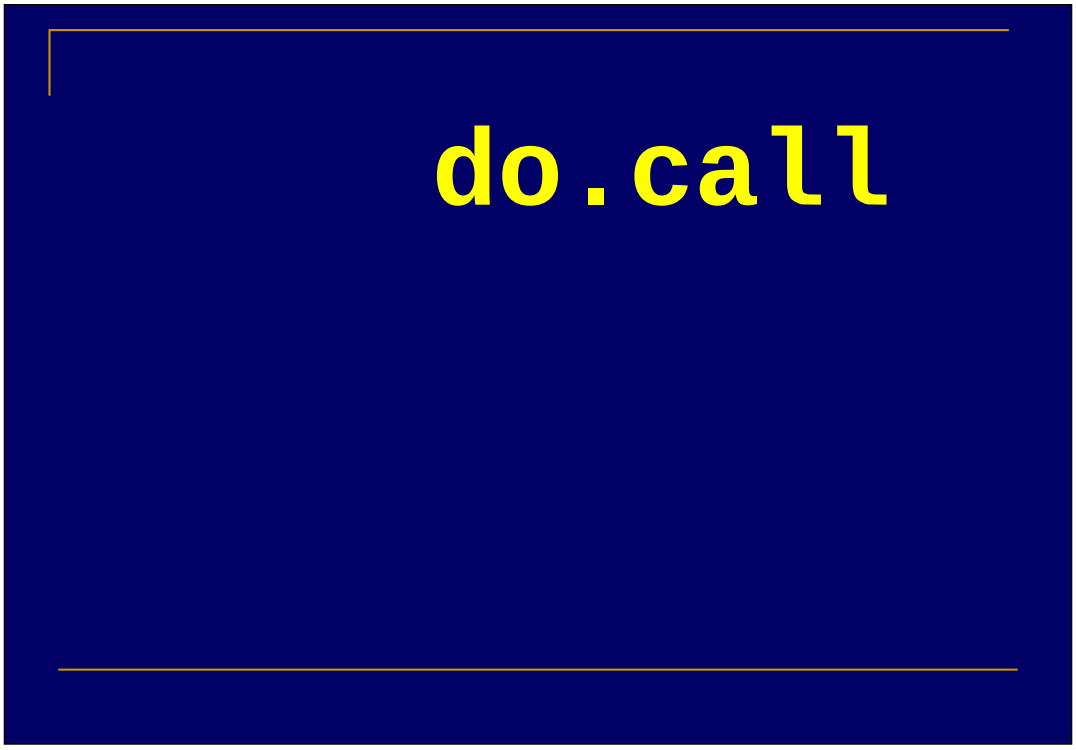
One alternative type of test is to create random inputs and then smell for smoke.



There are three steps in doing this.

If the code you are testing is not R code, then R is still a great place to do the first step of creating the random inputs.

If you are testing R code, then there are two key functions in performing the second step of calling the code.



If you don’t know this function, you should.

It allows you to call a function with an actual list of arguments.

So in the present case, we create a list containing the random inputs that were created in step 1, and then call the function being tested.

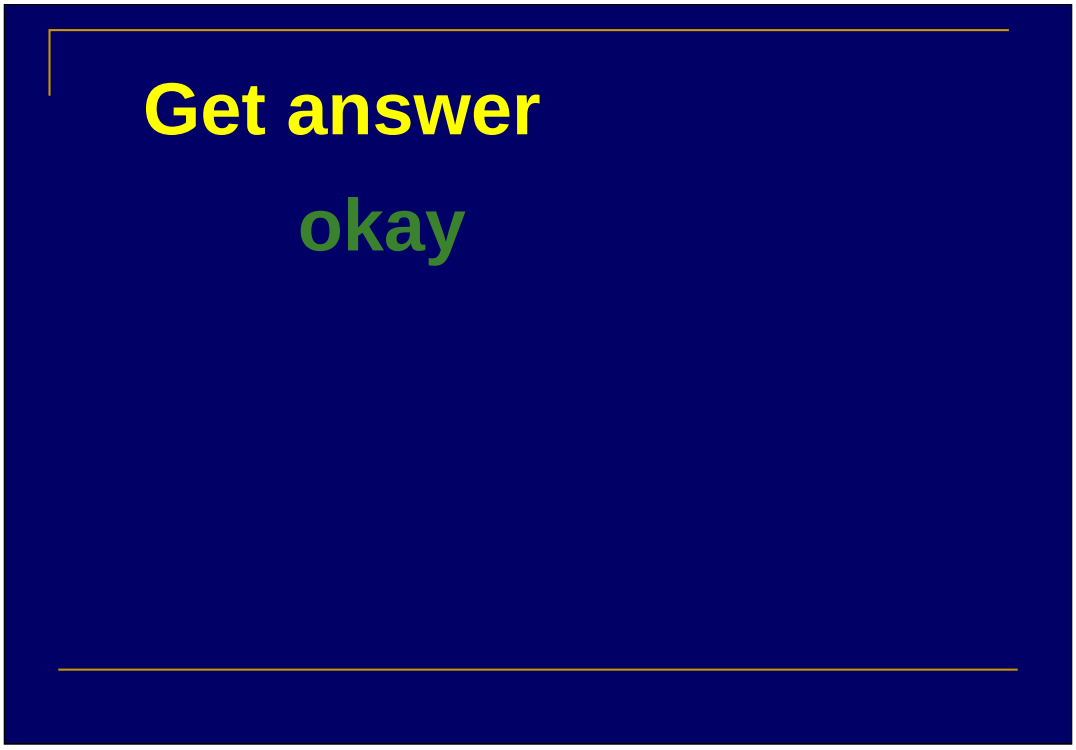
These are slightly different ways of doing the same thing.

They catch the errors from the test cases so that the errors don’t leak out into the actual test suite.

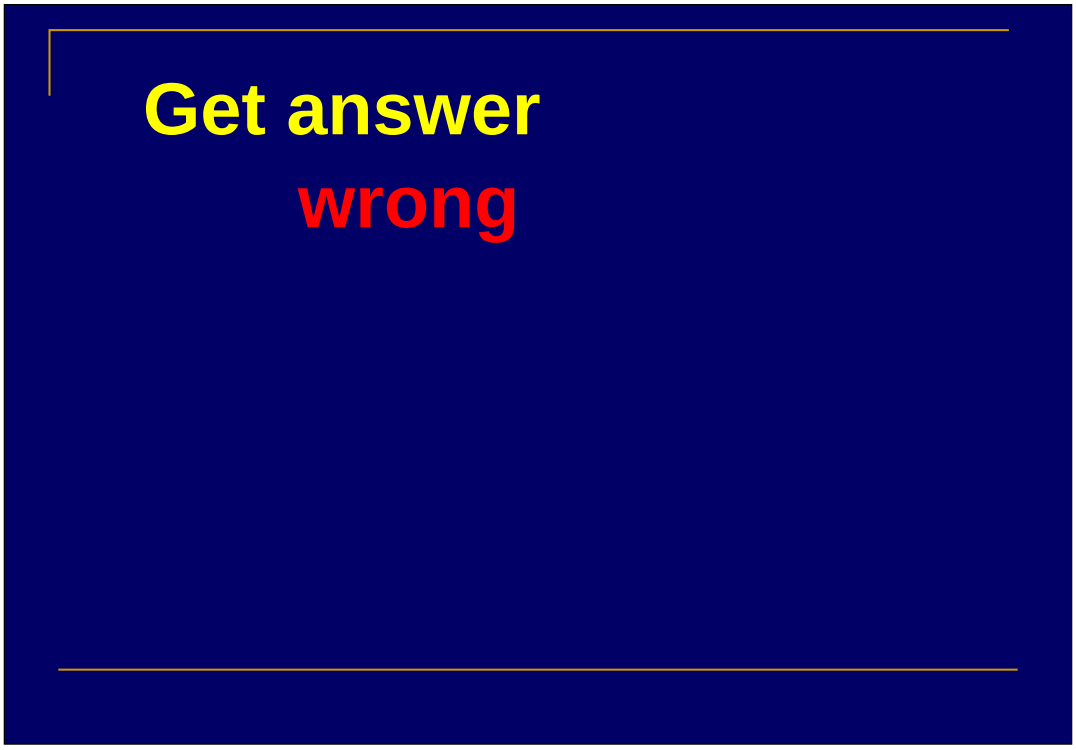
When doing random testing, there are several things that can happen. Here is the tree of those things.

Get answer

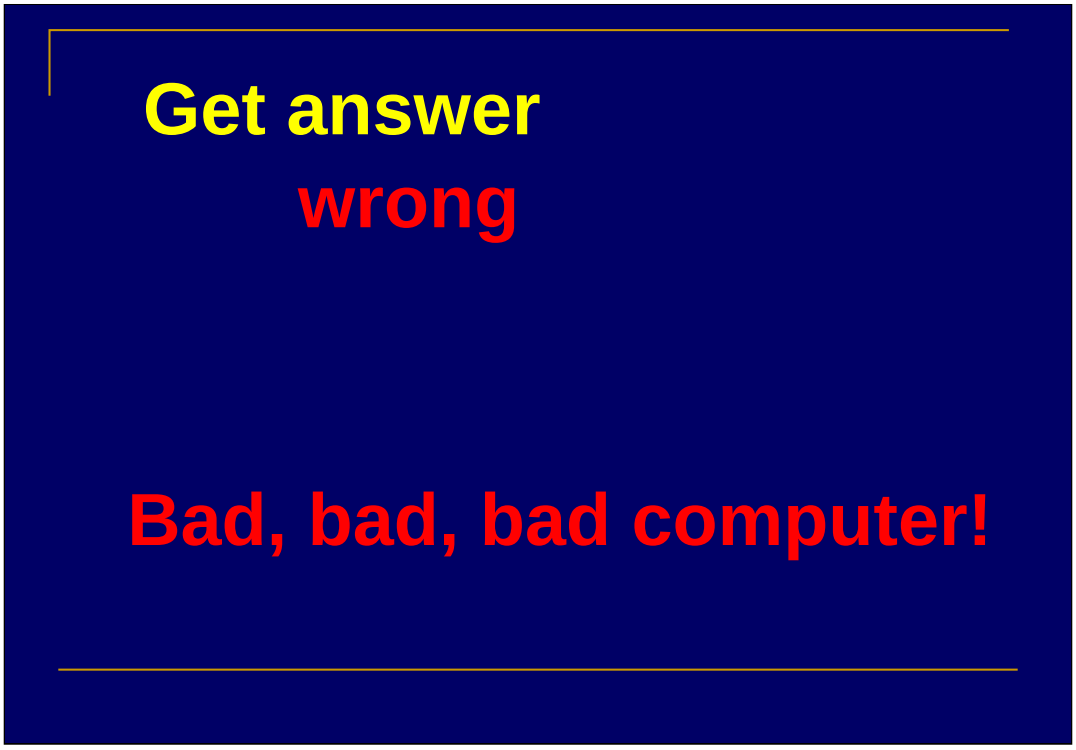
We can get an answer as opposed to an error.



Boring.



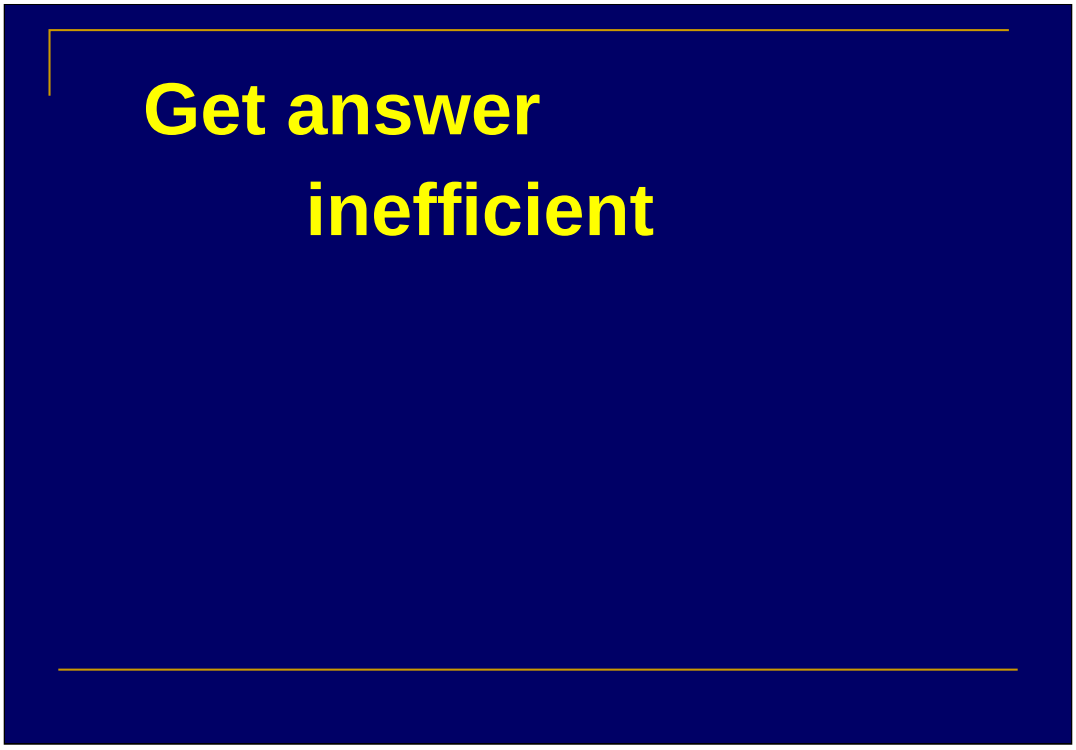
Not boring.



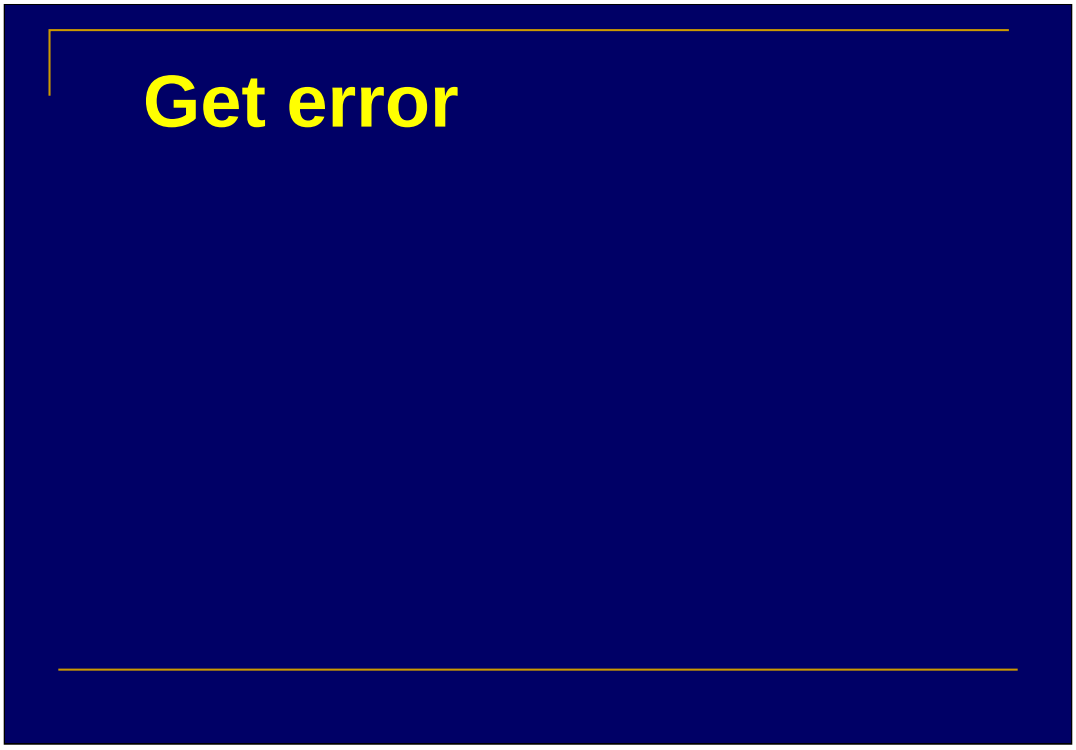
We don't like to be here. But it does make the effort of testing seem worthwhile.

If we are here, then we want to fix the problem. There are two more things we should do as well:

1. Create a case to put in the standard test suite for this problem. Random testing is an adjunct to standard testing, not a replacement.
2. Look through the code for other occurrences of this same sort of problem.



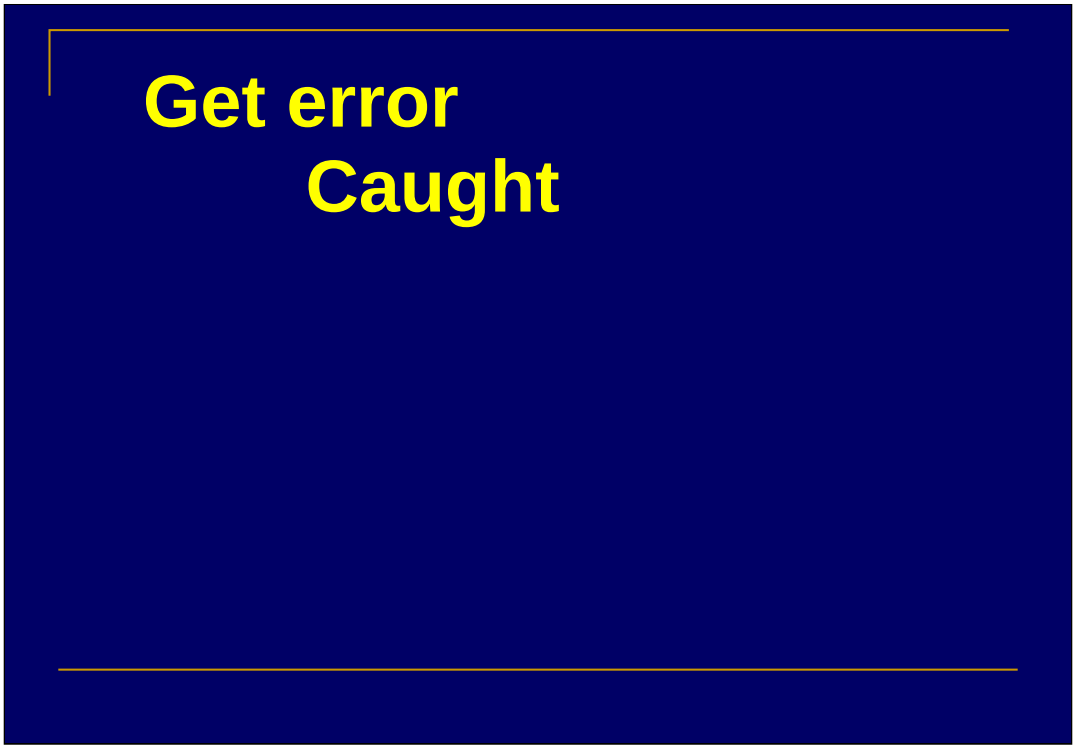
We can get the right answer via an inefficient (in time or memory) route. In my current application I'm not going to know this, but in other applications it is feasible to spot this.



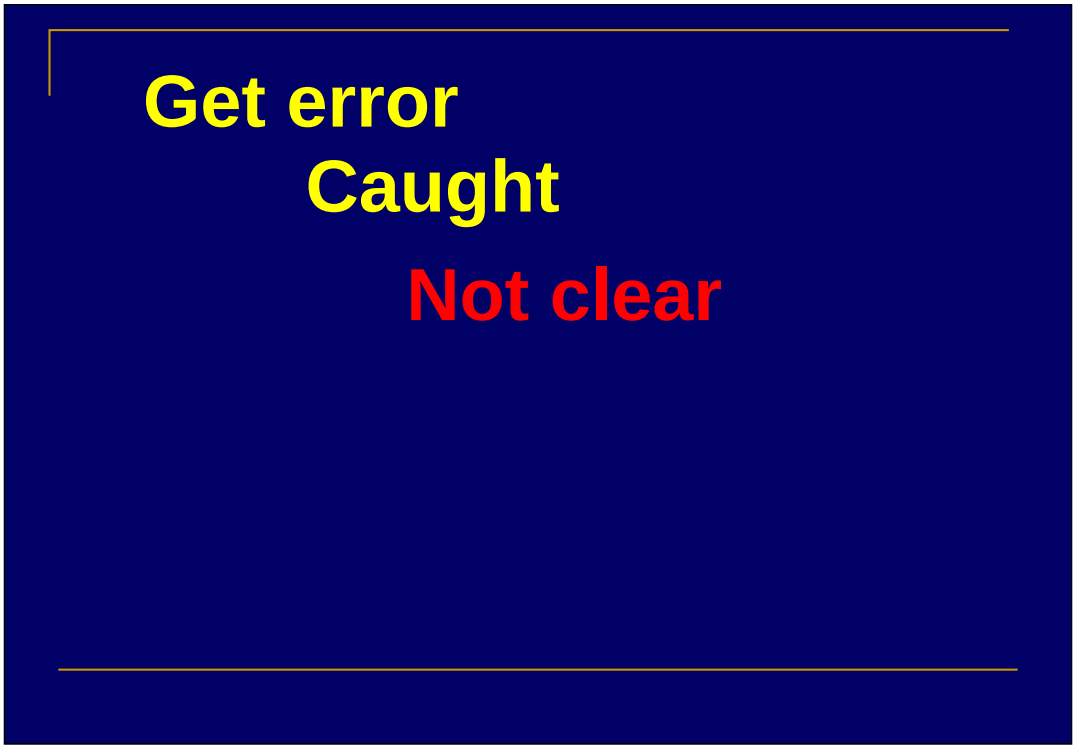
We can get an error rather than an answer.

This is a wonderful reason to do random testing. Standard test suites seldom to never exercise the code that throws errors. Random testing can exercise it a lot.

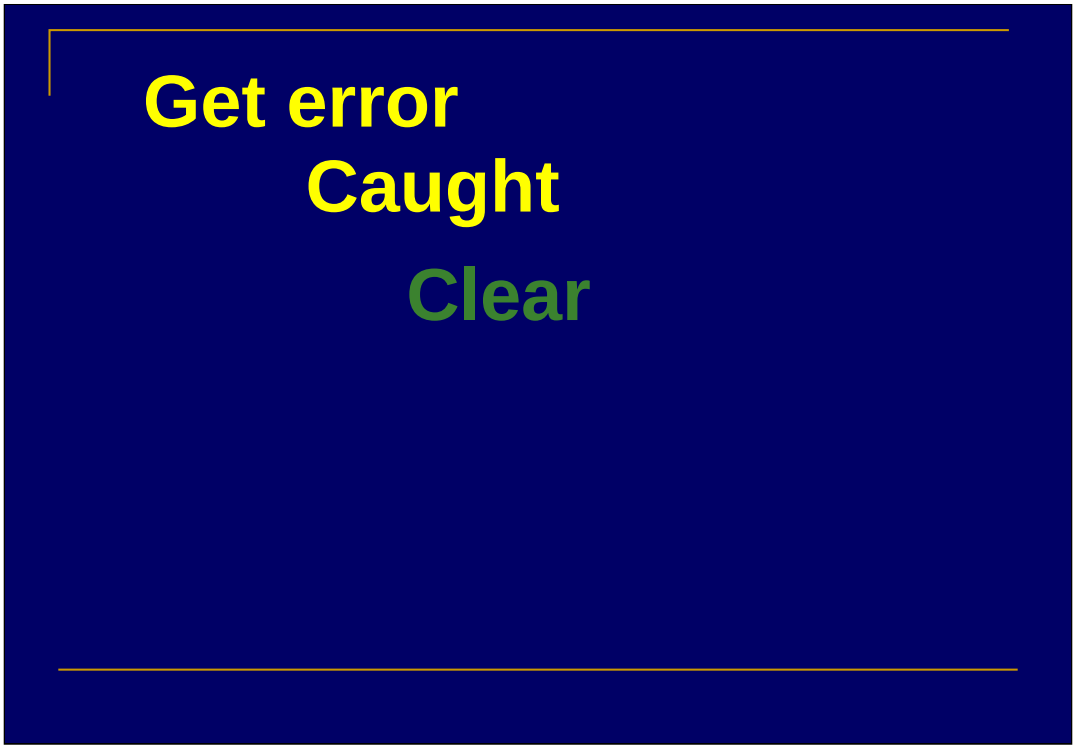
You could create a test where you don't even expect to get an answer – you're just testing the error throwing. This, of course, is entirely divorced from my original motivation for this sort of testing.



When we get an error, it can be caught by the code we've written.



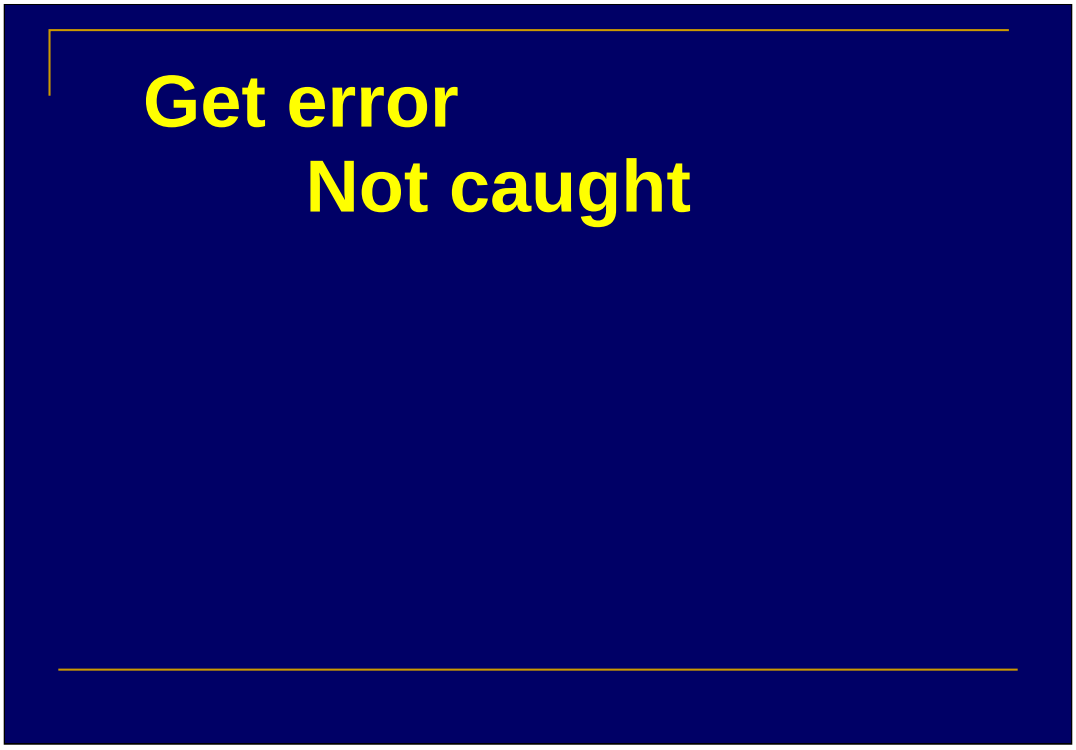
If the error message is not clear, then we should change the error message.



If the error message is clear, then we should question that assumption.

When we are coding, we have the curse of knowledge: we know everything that is going on. So any error message that we write is going to make sense to us.

We should try to put ourselves in the frame of mind of a user who is not especially clued in.

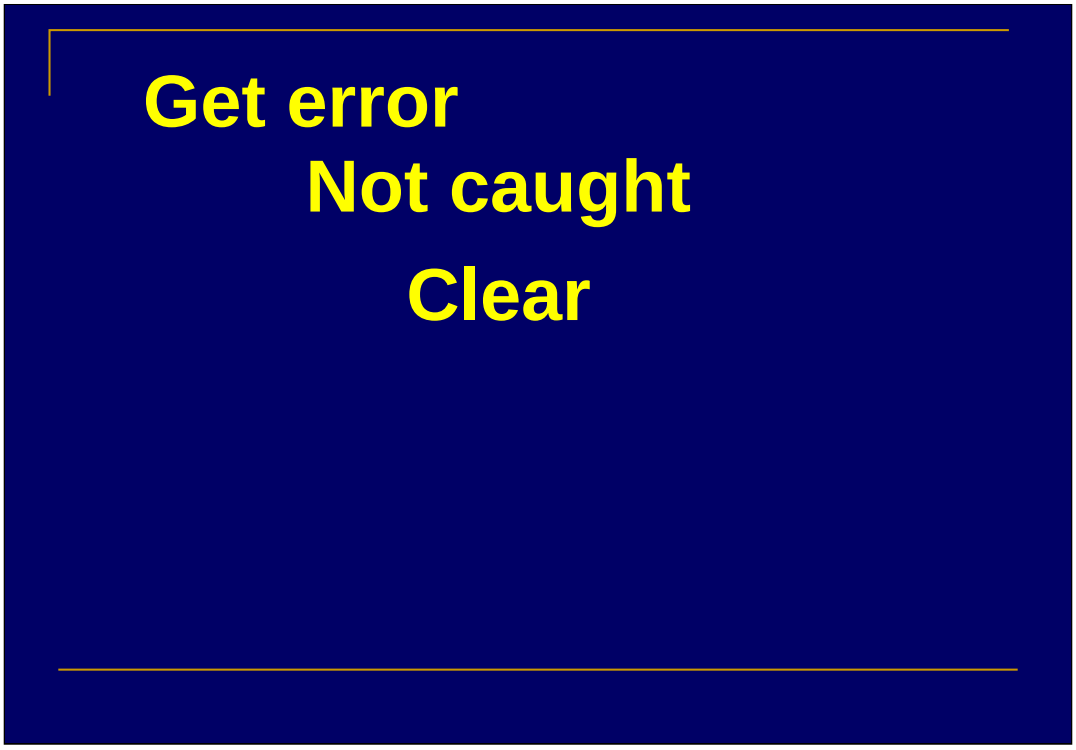


If the error message is not caught by our code,

Get error

Not caught

Not clear



we should consider catching it because it is even harder in that case for the message to be clear to our not-so-clued-in user.

I find writing test suites to be as exciting as digging ditches.

Of all the tasks involved in creating software, this is the lowest of the low for me.

I had expected the same to be true for random testing.

Photo by KayPat via stock.xchng

But I found random testing to have a quite different ambience to it. It is much more like playing.

Standard tests are basically static – write them once, periodically add bits to them, use them many times.

Random testing is more like an interactive game: do a bit, see how it goes, do some more, see how it goes, …

Keeping track of the different versions of the random tests is a good idea. You can go back and use a primitive test on a new version of the software. You can also use a primitive version as the basis for going off in a different testing direction.

I hope to see you on the playground.

Photo by forst747 via stock.xchng

One of the good things about giving a talk is that the audience tells you the name for what you are doing. “Fuzz testing” is a common word for random input testing.

One of the key questions is what distribution to use. The distribution will depend on your goal. It seems like there are three basic goals:

1. Test the correctness of answers
2. Test error throwing code
3. Test for security issues

The last of these is most closely associated with “fuzz testing”.

If your goal is one of the latter two, then the distributions should be very wide. That is, lots of stuff that need not even make any sense at all.

If you are trying to test the error throwing, then you probably want a more refined test as well that has arguments closer to what is expected.

If your goal is to test answers, then it seems to me that there are two possibilities:

- A distribution aiming at what is likely to be done in practice
- A distribution aiming at most likely to generate bugs

The reason that random testing is more fun is because it is really programming – writing the code that creates the distributions.

In my tests I have them start by doing two things:

- Print the set-seed number so that results are reproducible
- Print the version of the test being used

My tests so far have used the iteration variable as the set-seed number. Martin Maechler had a better idea: generate a random set-seed number. That way there is no need to keep track of what iterations have already been done with each test.

The business end of my random test function looks like:

```
ans <- try(do.call(“trade.optimizer”,
  c(thisdata, extra.args)))
if(inherits(ans, “try-error”)) {
     cat(“\n error with seed”, the.seed, “\n\n”)
} else {
     pprobe.check(ans)
}
```