What are the downsides using random values in Unit Testing?

Asked 14 years, 4 months ago Modified 5 years, 5 months ago

Viewed 10k times

34

I'm talking about a large scales system, with many servers and non deterministic input in high capacity. When i say non deterministic i'm talking about messages that are sent and you catch what you can and do the best you can. There are many types of messages, so the input could be very complicated. I can't imagine writing the code for so many scenarios and a simple non random (deterministic) message's generator is not good enough.

That's why i want to have a randomized unittest or server test that in case of a failure could write a log.

And i prefer the unittest instead of a random injector because i want it to run as part of the night build automated tests.

Any downsides?

unit-testing

Share

Improve this question

edited Jul 10, 2019 at 5:51

user

5,686 ● 9 ● 50 ● 79

Follow

asked Aug 9, 2010 at 15:42

Adibe7

3,539 ● 7 ● 32 ● 38

1 Note that if you seed the input of a random number generator (and only access it in a single thread) then the output is reproducable and not random. If the concern is just the generation of a large representative dataset, you can still use a random number generator without the test being random... – Kendrick Aug 9, 2010 at 18:08

11 Answers

Sorted by: Highest score (default)

48

Downsides

Firstly, it makes the test more convoluted and slightly harder to debug, as you cannot directly see all the values being fed in (though there's always the option of generating test cases as either code or data, too). If you're doing some semi-complicated logic to generate your random test data, then there's also the chance that this code has a bug in it. Bugs in test code can be a pain, especially if developers immediate assume the bug is the production code.

Secondly, it is often impossible to be specific about the expected answer. If you know the answer based on the input, then there's a decent chance you're just aping the logic under test (think about it -- if the input is random, how do you know the expected output?) As a result, you

may have to trade very specific asserts (the value should be x) for more general sanity-check asserts (the value should be between y and z).

Thirdly, unless there's a wide range of inputs and outputs, you can often cover the same range using well chosen values in a standard unit tests with less complexity. E.g. pick the numbers -max, (-max + 1), -2, -1, 0, 1, 2, max-1, max. (or whatever is interesting for the algorithm).

Upsides

When done well with the correct target, these tests can provide a very valuable complementary testing pass. I've seen quite a few bits of code that, when hammered by randomly generated test inputs, buckled due to unforeseen edge cases. I sometimes add an extra integration testing pass that generates a shedload of test cases.

Additional tricks

If one of your random tests fails, isolate the 'interesting' value and promote it into a standalone unit test to ensure that you can fix the bug and it will never regress prior to checkin.

edited Aug 11, 2010 at 10:54

answered Aug 9, 2010 at 16:29

Mark Simpson
23.4k ● 2 ● 46 ● 44

1 +1 I think the problem with the other answers is that it assumes you know what the edge cases are going to be. Sometimes a seemingly normal input can in fact be a very rare edge case. I'd give a 2nd +1 for promoting interesting values to standalone test if I could. – David Ly May 10, 2011 at 18:05

13

They are random.

(Your test might randomly work, even if your code is broken.)

Share Improve this answer

Follow

edited Jul 10, 2019 at 5:59

user
5,686 ● 9 ● 50 ● 79

answered Aug 9, 2010 at 15:44

relet
6,989 ● 2 ● 34 ● 42

1 i'm talking about additional testing to the non random tests – Adibe7 Aug 9, 2010 at 15:46

4 Your unit test is meant to prove, in a reproducible manner, if your unit works or not. If you do that in your other tests, you will not need an additional random test. You can use random values to scan your software for unknown security issues using random input. However with the result of that you should again write a reproducible unit test. – relet Aug 9, 2010 at 15:51

Tnx for your answer, can you take another look now, after i added some more explenations? – Adibe7 Aug 9, 2010 at 17:45

8

Also, you won't be able to reproduce tests many times over. A unit test should run exactly the same with given parameters.

Share Improve this answer

Follow

answered Aug 9, 2010 at 15:45

fernferret

856 ● 2 ● 13 ● 26

1 Thats not always true, random inputs can increase code coverage in some cases. – Nick Larsen Aug 9, 2010 at 15:49

3 @NIckLarsen: They may *happen* to increase code coverage, for one particular test run... You cannot *rely* on it covering that code though. You've gotta open the box to know if the cat is dead or not. – Pete Aug 9, 2010 at 15:55

1 @Nick - Then you should promote the value that increases code coverage. – Michael Lloyd Lee mlk Aug 11, 2010 at 10:15

2 While I agree with you guys, sometimes there are just too many cases to test all of them. Even if you limit it to exceptional cases, some systems just have too many. In those cases, testing against random inputs increases code coverage. The alternative in those situations is to prove your system for all classes of inputs. – Nick Larsen Aug 11, 2010 at 15:54

@pete: I'm not sure what you mean by `You've gotta open the box to know if the cat is dead or not` but hopefully you are not saying that implementation details should be tested! – Nick Larsen Aug 11, 2010 at 15:57

5

Randomizing unit tests is using a screwdriver to hammer in a nail. The problem is not that screwdrivers are bad; the problem is you're using the wrong tool for the job. The point of unit tests is to provide immediate feedback when you break something, so you can fix it right there.

Let's say you commit a change, which we'll call BadChange. BadChange introduces a bug, which your random tests will sometimes catch and sometimes not. This time, the tests don't catch it. BadChange is given the all-clear and goes into the code base.

Later, someone commits another change, GoodChange. GoodChange is one hundred percent fine. But this time, your random tests catch the bug introduced by BadChange. Now GoodChange is flagged as a problem, and the developer who wrote it will be going in circles trying to figure out why this innocuous change is causing issues.

Randomized testing is useful to constantly probe the whole application for issues, not to validate individual changes. It should live in a separate suite, and runs should not be linked to code changes; even if no one has made a change, the possibility remains that the random

tests will stumble across some exotic bug that previous runs missed.

Share Improve this answer

Follow

answered Oct 31, 2016 at 18:04

Evan Grantham-Brown

138 ● 2 ● 4

3

It is much better to have unit tests that are 100% repeatable and include all the edge cases. For example, test zero, negatives, positives, numbers too big, numbers too small, etc. If you want to include tests with random values in addition to all the edge cases and normal cases that would be fine. However, I'm not sure you would get much benefit out of the time spent. Having all the normal cases and edge cases should handle everything. The rest is "gravy".

Share Improve this answer

Follow

answered Aug 9, 2010 at 15:51

mpenrow

5,653 ● 9 ● 32 ● 36

This answer is simplistic. In the case of strings, for instance, you might not be able to represent the entire range (string of length 0, strings with invalid chars, really long strings). Some randomization will eventually discover some unforeseen edge case. – André Caron Jul 16, 2011 at 16:32

The results aren't repeatable, and depending on your tests, you may not know the specific conditions which caused the code to fail (thus making it tough to debug).

2

Share Improve this answer Follow

edited Jul 18, 2011 at 17:25

answered Aug 9, 2010 at 15:45

Kendrick

3,787 ● 1 ● 24 ● 42

3 That depends on the unit test framework. A decent test framework will output what values (and which assertion) caused the test to fail. This output can be used to build a new hard coded test case. – André Caron Jul 16, 2011 at 16:30

1

As others have suggested, it makes your test unreliable because you don't know what's going on inside of it. That means it might work for some cases, and not for others.

If you already have an idea of the range of values that you want to test, then you should either (1) create a different test for each value in the range, or (2) loop over the set of values and make an assertion on each iteration. A quick, rather silly, example...

```
for($i = 0; $i < 10; $i++)
  $this->assertEquals($i + 1, Math::addOne($i));
```

You could do something similar with character encodings. For example, loop over the ASCII character set and test all of those crazy characters against one of your text manipulation functions.

Share Improve this answer

Follow

answered Aug 9, 2010 at 15:52

Sam Bisbee

4,441 ● 22 ● 25

1

Upsides: they reveal when your other tests have *not* covered all the invariants. Whether you want your CI server to run nondeterministic tests is another issue. Given how incredibly useful I've found https://www.artima.com/shop/scalacheck, I don't intend to do without it from now on. Let's say you're implementing a pattern-matching algorithm. Do you really know all the different corner cases? I don't. Randomized inputs may flush them out.

Share Improve this answer

Follow

answered Oct 7, 2017 at 14:18

AbuNassar

1,215 ● 13 ● 12

0

You need to remember which random numbers you generated during verification.

Example.

```
Username= "username".rand();
Save_in_DB("user",Username); // To save it in DB
Verify_if_Saved("user",Username);
```

Share Improve this answer

Follow

answered Aug 9, 2010 at 16:15

Eastern Monk

6,605 ● 9 ● 48 ● 62

0

I believe that generating random input values can be a reliable testing technique when used together with equivalence partitioning. This means that, if you partition your input space and then randomly pick values from an equivalence class, then you are fine: same coverage (any of them, including statement, branch, all-uses etc). This under the assumption that your equivalence partitioning procedure is sound. Also, I would recommend boundary value analysis to be paired with equivalence partitioning and randomly generated inputs.

Finally, I would also recommend considering the TYPE of defects you want to detect: some testing techniques address specific types of defects, which might be hardly (and just by chance) detected by other techniques. An example: deadlock conditions.

In conclusion, I believe that generating random values is not a bad practice, in particular in some systems (e.g. web applications), but it only addresses a subset of existing defects (like any other technique) and one should be aware of that, so to complement his/her quality assurance process with the adequate set of activities.

Share Improve this answer Follow

edited Feb 20, 2013 at 3:49

answered Jan 17, 2013 at 19:59

Manu
4,137 ● 9 ● 52 ● 99

-1

Additional `Downside` that hasn't been mentioned as yet is that your tests may intermittently fail at random, especially when you randomly generate several test variables so they form a convoluted and sometimes untractable dependencies. See example here.

Debugging these is a right pain in the backside and sometimes is (next to) impossible.

Also, it's often hard to tell what your test actually tests (and if it tests anything at all).

Historically in my company we use random tests at multiple levels (Unit, Integration, SingleService Tests), and that seemed like a great idea initially - it saves you code, space and time allowing to test multiple scenarios in one test.

But increasingly that gets to be a sticky point in our development, when our (even historic and reliable in the past) test start failing at random - and fixing these is way labour-intensive.

Share Improve this answer

Follow

answered Jan 25, 2018 at 12:24

Nestor Milyaev

6,567 ● 4 ● 42 ● 55