Extracted from:

# Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do

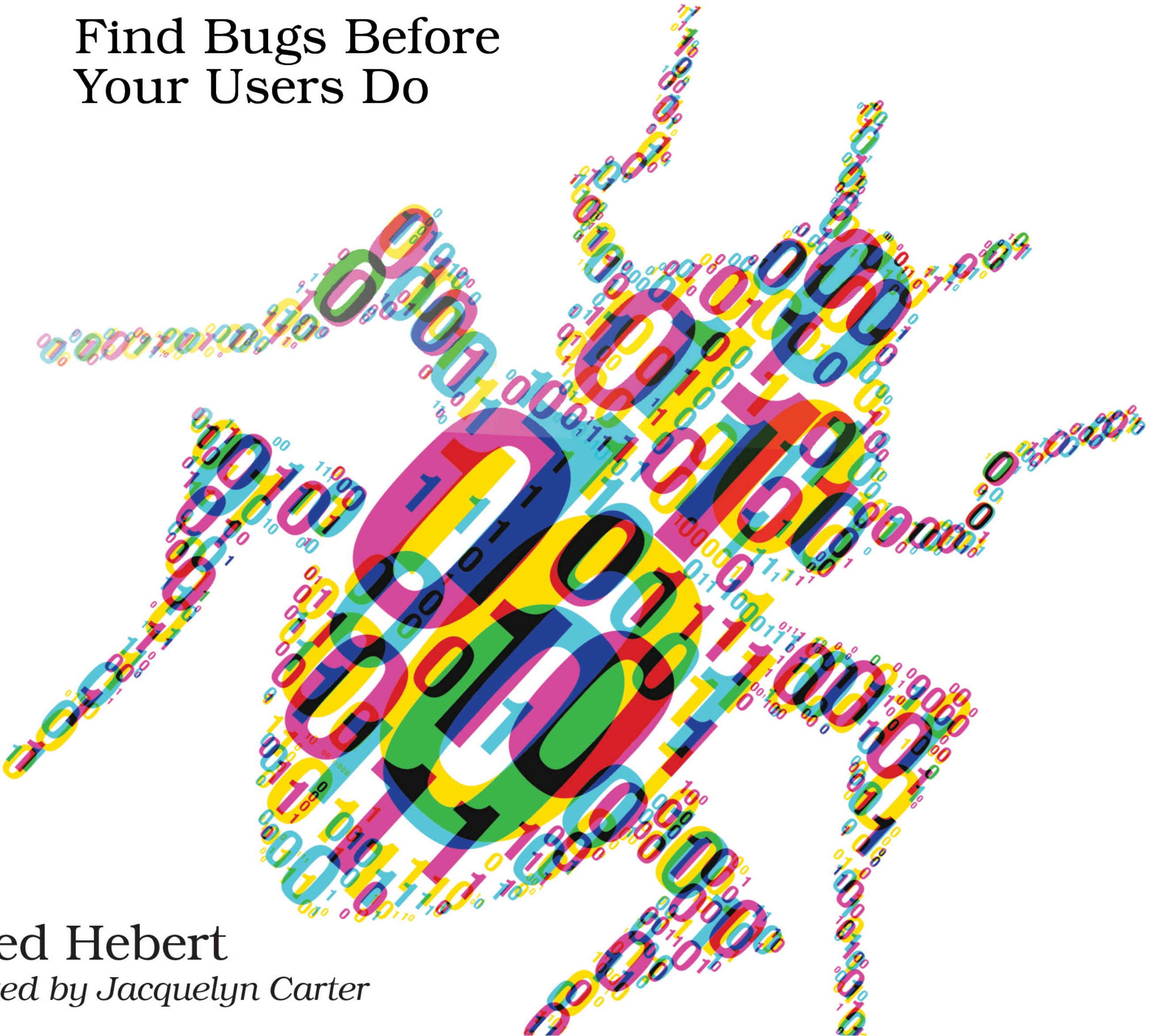
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do



Fred Hebert

*edited by Jacquelyn Carter*

# Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do

Fred Hebert

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Jacquelyn Carter
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

ISBN-13: 978-1-68050-621-1
Book version: P1.0—January 2019

CHAPTER 1

# Foundations of Property-Based Testing

Testing can be a pretty boring affair. It's a necessity you can't avoid: the one thing more annoying than having to write tests is not having tests for your code. Tests are critical to safe programs, especially those that change over time. They can also prove useful to help properly design programs, helping us write them as users as well as implementers. But mostly, tests are repetitive burdensome work.

Take a look at this example test that will check that an Erlang function can take a list of presorted lists and always return them merged as one single sorted list:

```
merge_test() ->
    [] = merge([]),
    [] = merge([[]]),
    [] = merge([[],[]]),
    [] = merge([[],[],[]]),
    [1] = merge([[1]]),
    [1,1,2,2] = merge([[1,2],[1,2]]),
    [1] = merge([[1],[],[]]),
    [1] = merge([[],[1],[]]),
    [1] = merge([[],[],[1]]),
    [1,2] = merge([[1],[2],[]]),
    [1,2] = merge([[1],[],[2]]),
    [1,2] = merge([[],[1],[2]]),
    [1,2,3,4,5,6] = merge([[1,2],[],[5,6],[],[3,4],[]]),
    [1,2,3,4] = merge([[4],[3],[2],[1]]),
    [1,2,3,4,5] = merge([[1],[2],[3],[4],[5]]),
    [1,2,3,4,5,6] = merge([[1],[2],[3],[4],[5],[6]]),
    [1,2,3,4,5,6,7,8,9] = merge([[1],[2],[3],[4],[5],[6],[7],[8],[9]]),
    Seq = seq(1,100),
    true = Seq == merge(map(fun(E) -> [E] end, Seq)),
    ok.
```

This is slightly modified code taken from the Erlang/OTP test suites for the lists module, one of the most central libraries in the entire language. It is boring, repetitive. It's the developer trying to think of all the possible ways the code can be used, and making sure that the result is predictable. You could probably think of another ten or thirty lines that could be added, and it could still be significant and explore the same code in somewhat different ways. Nevertheless, it's perfectly reasonable, usable, readable, and effective test code. The problem is that it's just so repetitive that a machine could do it. In fact, that's exactly the reason why traditional tests are boring. They're carefully laid out instructions where we tell the machine which test to run every time, with no variation, as a safety check.

It's not just that a machine *could* do it, it's that a machine *should* do it. We're spending our efforts the wrong way, and we could do better than this with the little time we have.

This is why property-based testing is one of the software development practices that generated the most excitement in the last few years. It promises better, more solid tests than nearly any other tool out there, with very little code. This means, similarly, that the software we develop with it should also get better—which is good, since overall software quality is fairly dreadful (including my own code). Property-based testing offers a lot of automation to keep the boring stuff away, but at the cost of a steeper learning curve and a lot of thinking to get it right, particularly when getting started. Here's what an equivalent property-based test could look like:

```
sorted_list(N) -> ?LET(L, list(N), sort(L)).

prop_merge() ->
    ?FORALL(List, list(sorted_list(pos_integer())),
        merge(List) == sort(append(List))).
```

Not only is this test shorter with just four lines of code, it covers more cases. In fact, it can cover hundreds of thousands of them. Right now the property-based test probably looks like a bunch of gibberish that can't be executed (at least not without the PropEr framework), but in due time, this will be pretty easy for you to read, and will possibly take less time to understand than even the long-form traditional test would.

Of course, not all test scenarios will be looking so favorable to property-based testing, but that is why you have this book. Let's get started right away: in this chapter, we'll see the results we should expect from property-based testing, and we'll cover the principles behind the practice and how they influence the way to write tests. We'll also pick the tools we need to get going, since, as you'll see, property-based testing does require a framework to be useful.

## Promises of Property-Based Testing

Property-based tests are different from traditional ones and require more thinking. A lot more. Good property-based testing is a learned and practiced skill, much like playing a musical instrument or getting good at drawing: you can get started and benefit from it in plenty of friendly ways, but the skill ceiling is very high. You could write property tests for years and still find ways to improve and get more out of them. Experts at property-based testing can do some pretty amazing stuff.

The good news is that even as a beginner, you can get good results out of property-based testing. You'll be able to write simple, short, and concise tests that automatically comb through your code the way only the most obsessive tester could. Your code coverage will go high and stay there even as you modify the program without changing the tests. You'll even be able to use these tests to find new edge cases without even needing to modify anything.

With a bit more experience, you'll be able to write straightforward integration tests of stateful systems that find complex and convoluted bugs nobody even thought could be hiding in there. In fact, if you currently feel like a code base without unit tests is not trustworthy, you'll probably discover the same phenomenon with property testing real soon. You'll have seen enough of what properties can do to see the shadows of bugs that haven't been discovered yet, and just feel something is missing, until you've given them a proper shakedown.

Overall, you'll see that property-based testing is not just using a bunch of tools to automate boring tasks, but a whole different way to approach testing, and to some extent, software design itself. Now that's a bold claim, but we only have to look at what experts can do to see why that might be so. An example can be found in Thomas Arts' slide set[1] and presentation[2] from the Erlang Factory 2016. In that talk, he mentioned using QuickCheck (the canonical property-based testing tool) to run tests on Project FIFO,[3] an open source cloud project. With a mere 460 lines of property tests, they covered 60,000 lines of production code and uncovered twenty-five important bugs, including:

- Timing errors
- Race conditions

1. http://www.erlang-factory.com/static/upload/media/1461230674757746pbterlangfactorypptx.pdf
2. https://youtu.be/iW2J7Of8jsE
3. https://project-fifo.net

- Type errors
- Incorrect use of library APIs
- Errors in documentation
- Errors in the program logic
- System limits errors
- Errors in fault handling
- One hardware error

Considering that some studies estimate that developers average six software faults per 1,000 lines of code, then finding twenty-five important bugs with 460 lines of tests is quite a feat. That's finding over fifty bugs per 1,000 lines of test, with each of these lines covering 140 lines of production code. If that doesn't make you want to become a pro, I don't know what will.

Let's take a look at some more-expert work. Joseph Wayne Norton ran a QuickCheck suite[4] of under 600 lines over Google's levelDB to find sequences of seventeen and thirty-one specific calls that could corrupt databases with ghost keys. No matter how dedicated to the task someone is, nobody would have found it easy to come up with the proper sequence of thirty-one calls required to corrupt a database.

Again, those are amazing results; that's a surprisingly low amount of code to find a high number of nontrivial errors on software that was otherwise already tested and running in production. Property-based testing is so impressive that it has wedged itself in multiple industries, including mission-critical telecommunication components,[5] databases,[6] components of cloud providers' routing and certificate-management layers, IoT platforms, and even in cars.[7]

We won't start at that level, but if you follow along (and do the exercises and practice enough), we might get pretty close. Hopefully, the early lessons will be enough for you to start applying property-based testing in your own projects. As we go, you'll probably feel like writing fewer and fewer traditional tests and will replace them with property tests instead. This is a really good thing, since you'll be able to delete code that you won't have to ever maintain again, at no loss in the quality of your software.

4. http://htmlpreview.github.io/?https://raw.github.com/strangeloop/lambdajam2013/master/slides/Norton-QuickCheck.html
5. http://www.erlang-factory.com/conference/ErlangUserConference2010/speakers/TorbenHoffman
6. http://www.erlang-factory.com/upload/presentations/255/RiakInside.pdf
7. http://www2015.taicpart.org/slides/Arts-TAICPART2015-Presentation.pdf

But as in playing music or drawing, things will sometimes be hard. When we hit a wall, it can be tempting (and easy) to tell ourselves that we just don't have that innate ability that experts must have. It's important to remember property-based testing is *not* a thing reserved for wizard programmers. The effort required to improve is continuous, and the progress is gradual and stepwise. Each wall you hit reveals an opportunity for improvement. We'll get there, one step at a time.

## Properties First

Before jumping to our tools and spitting out lines of code, the first thing to do is to get our fundamentals right and stop thinking property-based testing is about tests. It's about *properties*. Let's take a look at what the difference is, and what thinking in properties looks like.

Conceptually, properties are not that complex. Traditional tests are often example-based: you make a list of a bunch of inputs to a given program and then list the expected output. You may put in a couple of comments about what the code should do, but that's about it. Your test will be good if you can have examples that can exercise all the possible program states you have.

By comparison, property-based tests have nothing to do with listing examples by hand. Instead, you'll want to write some kind of meta test: you find a rule that dictates the behavior that should always be the same no matter what sample input you give to your program and encode that into some executable code—a *property*. A special test framework will then generate examples for you and run them against your property to check that the rule you came up with is respected by your program.

In short, traditional tests have you come up with examples that indirectly imply rules dictating the behavior of your code (if at all), and property-based testing asks you to come up with the rules first and then tests them for you. It's a similar distinction as the one you'd get between imperative and declarative programming styles, but pushed to the next level.

### The Example-Based Way

Here's an example. Let's say we have a function to represent a cash register. The function should take in a series of bills and coins representing what's already in the register, an amount of money due to be paid by the customer, and then the money handed by the customer to the cashier. It should return the bills and coins to cover the change due to the customer.

An approach based on unit tests may look like the following:

```
  %% Money in the cash register
  Register = [{20.00, 1}, {10.00, 2}, {5.00, 4},
              {1.00, 10}, {0.25, 10}, {0.01, 100}],
  %% Change                = cash(Register, PriceToPay, MoneyPaid),
❶ [{10.00, 1}]             = cash(Register, 10.00, 20.00),
❷ [{10.00, 1}, {0.25, 1}] = cash(Register,  9.75, 20.00),
  [{0.01, 18}]             = cash(Register,  0.82,  1.00),
❸ [{10.00, 1}, {5.00, 1}, {1.00, 3}, {0.25, 2}, {0.01, 13}]
                           = cash(Register,  1.37, 20.00).
```

At ❶, the test says that a customer paying a $10 item with $20 should expect a single $10 bill back. The case at ❷ says that for a $9.75 purchase paid with $20, a $10 bill with a quarter should be returned, for a total of $10.25. Finally, the test at ❸ shows a $1.37 item paid with a $20 bill yields $18.63 in change, with the specific cuts shown.

That's a fairly familiar approach. Come up with a bunch of arguments with which to call the function, do some thinking, and then write down the expected result. By listing many examples, we try to cover the full set of rules and edge cases that describe what the code should do. In property-based testing, we have to flip that around and come up with the rules first.

## The Properties-Based Way

The difficult part is figuring out how to go from our abstract ideas about the program behavior to specific rules expressed as code. Continuing with our cash register example, two rules to encode as properties could be:

- The amount of change is always going to add up to the amount paid minus the price charged.
- The bills and coins handed back for change are going to start from the biggest bill possible first, down to the smallest coin possible. This could alternatively be defined as trying to hand the customer as few individual pieces of money as possible.

Let's assume we magically encode them into functioning Erlang code (doing this for real is what this book is about—we can't show it all in the first chapter). Our test, as a property, could look something like this:

```
for_all(RegisterMoney, PriceToPay, MoneyPaid) ->
    Change = cash(RegisterMoney, PriceToPay, MoneyPaid),
    sum(Change) == MoneyPaid - PriceToPay
    and
    fewest_pieces_possible(RegisterMoney, Change).
```

Given some amount of money in the register, a price to pay, and an amount of money given by the customer, call the cash/3 function, and then check the change given. You can see the two rules encoded there: the first one checking that the change balances, and the second one checking that the smallest amount of bills and coins possible is returned (the implementation of which is not shown).

This property alone is useless. What is needed to make it functional is a property-based testing framework. The framework should figure out how to generate all the inputs required (RegisterMoney, PriceToPay, and MoneyPaid), and then it should run the property against all the inputs it has generated. If the property always remains true, the test is considered successful. If one of the test cases fails, a good property-based testing framework will modify the generated input until it can come up with one that can still provoke the failure, but that is as small as possible—a process called *shrinking*. Maybe it would find a mistake with a cash register that has a billion coins and bills in it, and an order price in the hundreds of thousands of dollars, but could then replicate the same failure with only $5 and a cheap item. If so, our debugging job is made a lot easier by the framework because a simple input means it's way easier to walk the code to figure out what happened.

For example, such a framework could generate inputs giving a call like cash([{1.00, 2}], 1.00, 2.00). Whatever the denomination, we might expect the cash/3 function would return a $1 bill and pass. Sooner or later, it would generate an input such as cash([{5.00, 1}], 20.00, 30.00), and then the program would crash and fail the property because there's not enough change in the register. Paying a $20 purchase with $30, even if the register holds only $5 is entirely possible: take $10 from the $30 and give it back to the customer. Is that specific amount possible to give back though? In real life, yes. We do it all the time. But in our program, since the money taken from the customer does not come in as bills, coins, or any specific denomination, there is no way to use part of the input money to form the output. The interface chosen for this function is wrong, and so are our tests.

## Comparing Approaches

Let's take a step back and compare example-based tests with property tests for this specific problem. With the former, even if all the examples we had come up with looked reasonable, we easily found ourselves working within the confines of the code and interface we had established. We were not really testing the code, we were describing its design, making sure it conformed to expectations and demands while making sure we don't slip in the future. This is valuable for sure, but properties gave us something more: they highlighted a failure of imagination.

Example-based unit tests made it easy to lock down bugs we could see coming, but those we couldn't see coming at all were left in place and would probably have made it to production. With properties (and a framework to execute them), we can instead explore the problem space more in depth, and find bugs and design issues much earlier. The math is simple: bugs that make it to production are by definition bugs that we couldn't see coming. If a tool lets us find them earlier, then production will see fewer bugs.

To put it another way, if example-based testing helps ensure that code does what we expect, property-based testing forces the exploration of the program's behavior to see what it can or cannot do, helping find whether our expectations were even right to begin with. In fact, when we test with properties, the design and growth of tests requires an equal part of growth and design of the program itself. A common pattern when a property fails will be to figure out if it's the system that is wrong or if it's our idea of what it should do that needs to change. We'll fix bugs, but we'll also fix our understanding of the problem space. We'll be surprised by how things we thought we knew are far more complex and tricky than we thought, and how often it happens.

That's thinking in properties.