Extracted from:

Property-Based Testing with PropEr, Erlang, and Elixir

Find Bugs Before Your Users Do

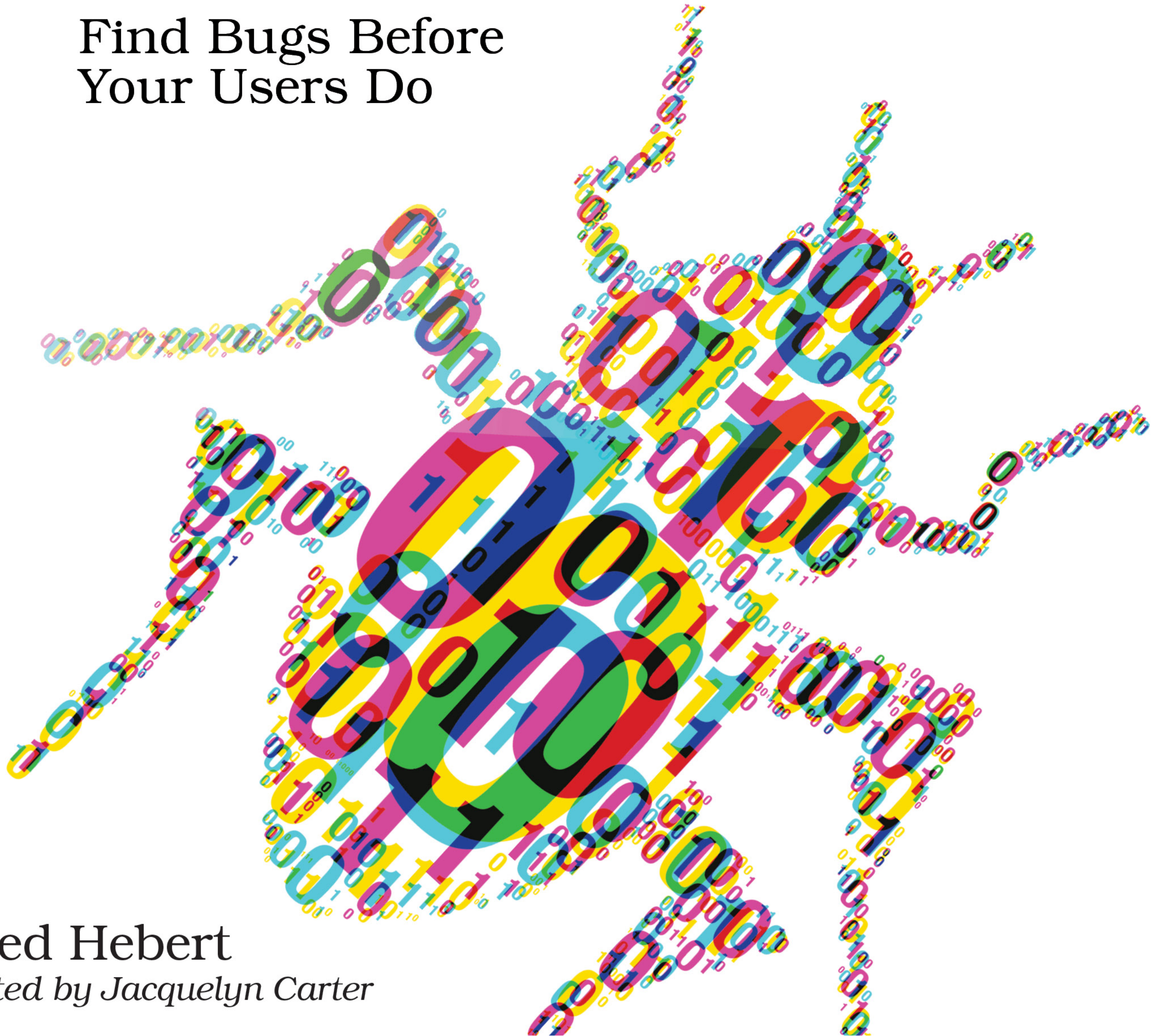
The Pragmatic Bookshelf

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

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

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

CHAPTER 3

Thinking in Properties

In the last chapter, we went over basic properties, including their syntax and generators that are available out of the box. We played with the `biggest(List)` function, ensuring it behaved properly. You may now have a good idea of what a property looks like and how to read it, but chances are you don't feel comfortable writing your own; it can take a long time to feel like you know how to write *effective* properties. All things considered, a big part of being good at coming up with properties is a question of experience: do it over and over again and keep trying your hand at it until it feels natural. That would usually take a long time, but we're going to try to speed things up.

Writing good properties is challenging and requires more effort than standard tests, but this chapter should provide some help. We're going to go through techniques that make the transition from using standard tests to thinking in properties feel natural. You'll become more efficient at progressing from having vague ideas of what your program should do to knowing how it should behave through well-defined properties.

We'll go over a few tips and tricks to help us figure out how to write decent enough properties in tricky situations. First, we'll try *modeling* our code, so we can skip over a lot of the challenging thinking that would otherwise be required. When that doesn't work, we'll try generalizing properties out of traditional example-based cases, which will help us determine the rules underpinning our expectations. Another approach we'll use is finding invariants so that we can ratchet up from trivial properties into a solid test suite. Finally, we'll implement symmetric properties as a kind of easy cheat code for some specific problems.

Modeling

Modeling essentially requires you to write an indirect and very simple implementation of your code—often an algorithmically inefficient one—and pit it against the real implementation. The model should be so simple that it is obviously correct. You can then optimize the real system as much as you want: as long as both implementations behave the same way, there's a good chance that the complex one is as good as the obviously correct one, but faster. So for code that does a conceptually simple thing, modeling is useful.

Let's revisit the biggest/1 function from last chapter and put it in its own module:

About the Code

In the snippets that follow, code is labeled with both the language (Erlang and Elixir) and the file where you should put the code if you're following along.

Erlang code/ThinkingInProperties/erlang/pbt/src/thinking.erl

```erlang
-module(thinking).
-export([biggest/1]).

biggest([Head | Tail]) ->
    biggest(Tail, Head).

biggest([], Biggest) ->
    Biggest;
biggest([Head|Tail], Biggest) when Head >= Biggest ->
    biggest(Tail, Head);
biggest([Head|Tail], Biggest) when Head < Biggest ->
    biggest(Tail, Biggest).
```

Elixir code/ThinkingInProperties/elixir/pbt/lib/pbt.ex

```elixir
defmodule Pbt do
  def biggest([head | tail]) do
    biggest(tail, head)
  end

  defp biggest([], max) do
    max
  end

  defp biggest([head | tail], max) when head >= max do
    biggest(tail, head)
  end

  defp biggest([head | tail], max) when head < max do
    biggest(tail, max)
  end

end
```

The function iterates over the list in a single pass: the largest value seen is held in memory and replaced any time a larger one is spotted. Once the list has been fully scanned, the largest value seen so far is also the largest value of the list. That value is returned, and everything is fine.

The challenge is coming up with a good property for it. The obvious rule to encode is that the function should return the biggest number of the list passed in. The problem with this obvious rule is that it's hard to encode: the biggest/1 function is so simple and such a direct implementation of the rule that it's hard to make a property that is not going to be a copy of the function itself. Doing so would not be valuable, because we're likely to repeat the same mistakes in both places, so we might as well not test it.

In these cases, modeling is a good idea. So for this function, we need to come up with an alternative implementation that we can trust to be correct to make the property work. Here's the property for this case:

```
Erlang          code/ThinkingInProperties/erlang/pbt/test/prop_thinking.erl
```

```erlang
-module(prop_thinking).
-include_lib("proper/include/proper.hrl").

%%%%%%%%%%%%%%%%%%
%%% Properties %%%
%%%%%%%%%%%%%%%%%%

prop_biggest() ->
    ?FORALL(List, non_empty(list(integer())),
        begin
            thinking:biggest(List) =:= model_biggest(List)
        end).
```

```
Elixir               code/ThinkingInProperties/elixir/pbt/test/pbt_test.exs
```

```elixir
property "finds biggest element" do
  forall x <- non_empty(list(integer())) do
    Pbt.biggest(x) == model_biggest(x)
  end
end
```

Most modeling approaches will look like that one. The crucial part is the model, represented by model_biggest/1. To implement the model, we can pick standard library functions to give us our alternative, slower, but so-simple-it-must-be-correct implementation:

```
Erlang
```

```erlang
model_biggest(List) ->
    lists:last(lists:sort(List)).
```

```
Elixir
```

```
def model_biggest(list) do
  List.last(Enum.sort(list))
end
```

Since sorting a list orders all its elements from the smallest one to the largest one possible, picking the last element of the list should logically give us the biggest list entry. Running the property shows that it is good enough:

```
$ rebar3 proper -p prop_biggest
===> Verifying dependencies...
===> Compiling pbt
===> Testing prop_thinking:prop_biggest()
...................................................................
.................................
OK: Passed 100 test(s).
===>
1/1 properties passed
```

Chances are pretty much null that all the functions involved are buggy enough that we can't trust our test to be useful and it only passes by accident. In fact, as a general rule when modeling, we can assume that our code implementation is going to be as reliable as the model to which we compare it. That's why you should always aim to have models so simple they are obviously correct. In the case of biggest/1, it's now as trustworthy as lists:sort/1 and lists:last/1.

Modeling is also useful for integration tests of stateful systems with lots of side effects or dependencies, where "how the system does something" is complex, but "what the user perceives" is simple. Real-world libraries or systems often hide such complexities from the user to appear useful at all. Databases, for example, can do a lot of fancy operations to maintain transactional semantics, avoid loss of data, and keep good performance, but a lot of these operations can be modeled with simple in-memory data structures accessed sequentially.

Finally, there's a rare but great type of modeling that may be available, called the *oracle*. An oracle is a reference implementation, possibly coming from a different language or software package, that can therefore act as a prewritten model. The only thing required to test the system is to compare your implementation with the oracle and ensure they behave the same way.

If you can find a way to model your program, you can get pretty reliable tests that are easy to understand. You have to be careful about performance—if your so-simple-it-is-correct implementation is dead slow, so will your tests be—but models are often a good way to get started.

Generalizing Example Tests

Modeling tends to work well, as long as it is possible to write the same program multiple times, and as long as one of the implementations is so simple it is obviously correct. This is not always practical, and sometimes not possible. We need to find better properties. That's significantly harder than finding *any* property, which can already prove difficult and requires a solid understanding of the problem space. A good trick to find a property is to just start by writing a regular unit test and then abstract it away. We can take the steps that are common to coming up with all individual examples and replace them with generators.

In the previous section, we said that biggest/1 is as reliable as lists:sort/1 and lists:last/1, the two functions we used to model it in its property. Our model's correctness entirely depends on these two functions doing the right thing. To make sure they're well-behaved, we'll write some tests demonstrating they work as expected. Let's see how we can write a property for lists:last/1. This function is so simple that we can consider it to be axiomatic—just assume it's correct—and a fundamental block of the system. For this kind of function, traditional unit tests are usually a good fit since it's easy to come up with examples that should be significant. We can also transform examples into a property. After all, if we can get a property to do the work for us, we'll have thousands of examples instead of the few we'd come up with, and that's objectively better coverage.

Let's take a look at what example tests could look like for lists:last/1, so that we can generalize them into a property:

```
last_test() ->
    ?assert(-23 =:= lists:last([-23])),
    ?assert(5 =:= lists:last([1,2,3,4,5])),
    ?assert(3 =:= lists:last([5,4,3])).
```

We can write this test by hand:

1. Construct a list by picking a bunch of numbers.
    - Pick a first number.
    - Pick a second number.
    - …
    - Pick a last number.
2. Take note of the last number in the list as the expected one.
3. Check that the value expected is the one obtained.

Since the last substep of *1.* (“Pick a last number.”) is the one we really want to focus on, we can break it from the other substeps by using some clever generator usage. If we group all of the initial substeps in a list and isolate the last one, we get something like {list(number()), number()}. Here it is used in a property:

```
Erlang          code/ThinkingInProperties/erlang/pbt/test/prop_thinking.erl
```

```erlang
prop_last() ->
    %% pick a list and a last number
    ?FORALL({List, KnownLast}, {list(number()), number()},
        begin
            KnownList = List ++ [KnownLast], % known number appended to list
            KnownLast =:= lists:last(KnownList) % known last number is found
        end).
```

```
Elixir               code/ThinkingInProperties/elixir/pbt/test/pbt_test.exs
```

```elixir
property "picks the last number" do
  forall {list, known_last} <- {list(number()), number()} do
    known_list = list ++ [known_last]
    known_last == List.last(known_list)
  end
end
```

And just like that, we’ll get hundreds or even millions of example cases instead of a few unit tests all done by hand. Of course, we now have to believe the ++ operator will correctly append items to a list if we want to trust this new property. We’re getting pulled in deeper: is it possible to make a model out of it? Can it be turned into a simpler property, or just tested with traditional unit tests? This is ultimately a question about which parts of the system you just trust to be correct, and that is left for you to decide. It’s challenging to write a lot of significant tests for very simple cases, but the next technique can help.

Invariants

Some programs and functions are complex to describe and reason about. They could be needing a ton of small parts to all work right for the whole to be correct, or we may not be able to assert their quality because it is just hard to define. For example, it’s hard to say why a meal is good, but it might include criteria like: the ingredients are cooked adequately, the food is hot enough, it’s not too salty, not too sweet, not too bitter, it’s well-presented, the portion size is reasonable, and so on. Those factors are all easier to measure objectively and can be a good proxy for “the customer will enjoy the food.” In a software system, we can identify similar conditions or facts that should always remain true. We call them *invariants*, and testing for them is a great way to get around the fact that things may just be ambiguous otherwise.

If an invariant were to be false at any time, you would know something is messed up. Seriously messed up. Here are some examples:

- A store cannot sell more items than it has in stock.
- In a binary search tree, the left child is smaller and the right child is greater than their parent's value.
- Once you insert a record in a database, you should be able to read it back and not see it as missing.

A single invariant on its own is usually not enough to show a piece of code is working as expected. But if we can come up with many invariants and small things to validate, and if they *all* always remain true, we can gain a lot more confidence in the ability of our code base to work well. Strong ropes are built from smaller threads put together. In papers or proofs about why a given data structure works, you'll find that almost all aspects of its success comes from ensuring a few invariants are respected.

For property-based testing, we can write a lot of simple properties, each representing one invariant. As we add more and more of them, we can build a strong test suite that overall demonstrates that our code is rock solid.

The `lists:sort/1` function is a good example of a piece of code that could be checked with invariants. How can we identify the invariants though? We could pick the first one by saying "a sorted list has all the numbers in it ordered from smallest to largest." The problem is that this is such a complete and accurate description of the whole function that if we used it as an invariant, we'd need a complete sorting function to test it. This is circular as it boils down to saying "a proper sort function is a function that sorts properly." A test that is written the same way as the code it tests is not useful.

Instead we should try to break it down into smaller parts. Something like "each number in a sorted list is smaller than (or equal to) the one that follows." The difference is small, but important. In the first case, we declare the final state of the entire list, the intended outcome. In the latter case, we mention an invariant that should be true of any pair of elements, and not the whole output. We can do an entirely local verification without having the whole picture. Then, when we apply the property to every pair, we indirectly test for a fully ordered output:

Erlang code/ThinkingInProperties/erlang/pbt/test/prop_thinking.erl

```erlang
prop_sort() ->
    ?FORALL(List, list(term()),
            is_ordered(lists:sort(List))).

is_ordered([A,B|T]) ->
    A =< B andalso is_ordered([B|T]);
is_ordered(_) -> % lists with fewer than 2 elements
    true.
```

Elixir code/ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```elixir
property "a sorted list has ordered pairs" do
  forall list <- list(term()) do
    is_ordered(Enum.sort(list))
  end
end

def is_ordered([a, b | t]) do
  a <= b and is_ordered([b | t])
end

# lists with fewer than 2 elements
def is_ordered(_) do
  true
end
```

Not bad. A good side effect of this approach is that the implementation is almost guaranteed to be different from the test: we only validated that some property held, and didn't transform the input at all. No modeling is involved here. As mentioned earlier though, a single invariant isn't very solid. If we'd written a sort function as follows, it would always pass:

```erlang
sort(_) -> [].
```

We need more invariants to ensure the implementation is right. We can look for other properties that should always be true and easy to check. Here are some examples:

- The sorted and unsorted lists should both have the same size.
- Any element in the sorted list has to have its equivalent in the unsorted list (no element added).
- Any element in the unsorted list has to have its equivalent in the sorted list (no element dropped).

Let’s see how these could be implemented:

Erlang code/ThinkingInProperties/erlang/pbt/test/prop_thinking.erl

```erlang
%% @doc the sorted and unsorted list should both remain the same size
prop_same_size() ->
    ?FORALL(L, list(number()),
            length(L) =:= length(lists:sort(L))).

%% @doc any element in the sorted list has to have its equivalent in
%% the unsorted list
prop_no_added() ->
    ?FORALL(L, list(number()),
        begin
            Sorted = lists:sort(L),
            lists:all(fun(Element) -> lists:member(Element, L) end, Sorted)
        end).

%% @doc any element in the unsorted list has to have its equivalent in
%% the sorted list
prop_no_removed() ->
    ?FORALL(L, list(number()),
        begin
            Sorted = lists:sort(L),
            lists:all(fun(Element) -> lists:member(Element, Sorted) end, L)
        end).
```

Elixir code/ThinkingInProperties/elixir/pbt/test/pbt_test.exs

```elixir
property "a sorted list keeps its size" do
  forall l <- list(number()) do
    length(l) == length(Enum.sort(l))
  end
end

property "no element added" do
  forall l <- list(number()) do
    sorted = Enum.sort(l)
    Enum.all?(sorted, fn element -> element in l end)
  end
end

property "no element deleted" do
  forall l <- list(number()) do
    sorted = Enum.sort(l)
    Enum.all?(l, fn element -> element in sorted end)
  end
end
```

That's better. Now it's harder to cheat your way through the properties, and we can trust our tests:

```
$ rebar3 proper
«build output»
===> Testing prop_sort:prop_sort()
........................................................................
............................
OK: Passed 100 test(s).
===> Testing prop_sort:prop_same_size()
........................................................................
............................
OK: Passed 100 test(s).
===> Testing prop_sort:prop_no_added()
........................................................................
............................
OK: Passed 100 test(s).
===> Testing prop_sort:prop_no_removed()
........................................................................
............................
OK: Passed 100 test(s).
===>
4/4 properties passed
```

Each of these properties is pretty simple on its own, but they make a solid suite against almost any sorting function. Another great aspect is that some invariants are easy to think about, are usually fast to validate, and are almost always going to be useful as a sanity check, no matter what. They will combine well with every other testing approach you can think of.

A small gotcha here is that our tests now depend on other functions from the lists module. This brings us back to the discussion on when to stop, since we need to trust these other functions if we want our own tests to be trustworthy. We could just call the shots and say we trust them, especially since they are given to us by the language designers. It's a calculated risk. But there's another interesting approach we could use by testing them all at once.