Don’t write tests; Generate them

Chris Keathley / @ChrisKeathley / c@keathley.io
What are we gunna talk about?
Testing today
What are property tests
Basic Example
Real world(ish) example
Why Elixir?
Elixir is a functional, dynamic language that targets the Erlang VM (BEAM)
Testing
Test Driven Development

1. Write a failing test
2. Write enough code to make that test pass
3. Refactor
Test Driven Development
Validation
Protection From Regression
Design
Happiness

quiet contemplation

Trough of disillusionment

Property Tests!!!

Career

TDD
Tests provide guard rails
Tests directly couple your implementation
Tests directly couple your implementation
Tests directly couple your implementation
Write as few tests as possible
Warning: Contrived Strawman Argument incoming!
Let's TDD Addition!

\[ x + y = ? \]
test "adding 2 numbers" do
end
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
end
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
end

def add(x, y) do
end
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
end

def add(x, y) do
  2
end
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
end

def add(_x, _y) do
  2
end
But wait...
Chris Keathley   3:14 PM
Hey there is an issue with the add function. It doesn't seem to work with \texttt{add(3, 4)}. 
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
end

def add(_x, _y) do
  2
end
Pattern Matching
Pattern Matching
Pattern Matching

\[ x = 3 \]
Pattern Matching

\[ x = 3 \]

\[ y = x \]
Pattern Matching

\[ x = 3 \]
\[ y = x \]
\[ 3 = y \]
Pattern Matching is an assertion
Pattern Matching

\[
\begin{align*}
X &= 3 \\
3 &= X
\end{align*}
\]
Pattern Matching

3 = 3
Pattern Matching

{%name: "Chris", hobbies: ["Coffee", "Pinball", "Lego"]%}
Pattern Matching

```ruby
{name: user_name} = %{name: "Chris", hobbies: ["Coffee", "Pinball", "Lego"]}
```
Pattern Matching

user_name = "Chris"
def user_name(user_map) do
  %{name: name} = user_map
  name
end
Pattern Matching

def user_name(%{name: name}) do
  name
end
def user_name(%{name: name}) do
  name
end

def user_name(_) do: "Default User"
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
end

def add(_x, _y) do
  2
end
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
end

def add(_x, _y), do: 2
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
end

def add(3, _), do: 7
def add(_x, _y), do: 2
keathley in ~/D/t/g/qc_example on master
But wait...
Hey found another edge case with `add`. Looks like it doesn't work if you try to add a negative value.
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
end

def add(3, _), do: 7
def add(_x, _y), do: 2
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
  assert add(-1, 4) == 3
end

def add(3, _), do: 7
def add(_x, _y), do: 2
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
  assert add(-1, 4) == 3
end

def add(3, _), do: 7
def add(_x, _y), do: 2
Guard clauses
Guard clauses

```ruby
def user_name(%{name: name}) do
  name
end
```
Guard clauses

```ruby
def user_name(%{name: name}) when is_binary(name) do
  name
end
```
Guard clauses

def user_name(%{name: name}) when is_binary(name) do
  name
end
def user_name(%{name: name, age: age}) when age < 20
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
  assert add(-1, 4) == 3
end

def add(3, _), do: 7
def add(_x, _y), do: 2
test "adding 2 numbers" do
  assert add(1, 1) == 2
  assert add(0, 2) == 2
  assert add(3, 4) == 7
  assert add(-1, 4) == 3
end

def add(x, _) when x < 0, do: 3
def add(3, _), do: 7
def add(_x, _y), do: 2
All checks have passed
1 successful check

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request
You can also open this in GitHub Desktop or view command line instructions.
Chris Keathley  9:03 PM
Looks great! I think we finally found all the bugs
What have we gained?
TDD
Validation
Protection From Regression
Design
TDD

Validation

Protection From Regression

Design
TDD

Protection From Regression

Design
TDD

Validation

Protection From Regression

Design ?
There are bugs in your code
Writing tests for one feature
Writing tests for one feature
Writing tests for two features

\(o(n)\)

\(o(n^2)\)
Writing tests for three features

$O(n^3)$

$O(n^2)$

$O(n)$
Race conditions?
Property tests
QuickCheck: 
A Lightweight Tool for Random Testing 
of Haskell Programs

Koen Claessen  
Chalmers University of Technology  
koen@cs.chalmers.se

John Hughes  
Chalmers University of Technology  
rjmh@cs.chalmers.se

ABSTRACT
QuickCheck is a tool which aids the Haskell programmer in formulating and testing properties of programs. Properties are described as Haskell functions, and can be automatically tested on random input, but it is also possible to define custom test data generators. We present a number of case studies, in which the tool was successfully used, and also point out some pitfalls to avoid. Random testing is especially suitable for functional programs because properties can be stated at a fine grain. When a function is built from separately tested components, then random testing suffices to obtain good coverage of the definition under test.

1. INTRODUCTION
Testing is by far the most commonly used approach to software development, and yet it is usually the most neglected. A common reason for this is that testing is often seen as a step at the end of the development process, rather than a fundamental part of it. Testing is also often seen as a burden, rather than an opportunity. However, testing is crucial for ensuring the quality of a software product. A well-tested program is more likely to be reliable, and is also more likely to be maintainable.

A testing tool must be able to determine whether a test is passed or failed; the human tester must supply an automatically checkable criterion of doing so. We have chosen to use formal specifications for this purpose. We have designed a simple domain-specific language of testable specifications which the tester uses to define expected properties of the functions under test. QuickCheck then checks that the properties hold in a large number of cases. The specification language is embedded in Haskell using the class system. Properties are normally written in the same module as the functions they test, where they serve also as checkable documentation of the behaviour of the code.

A testing tool must also be able to generate test cases automatically. We have chosen the simplest method, random
Expected == Actual
Expected == Actual

Overly Specific
Property Tests
Property Tests

Int
Property Tests

\[ \text{Int} \rightarrow p(x) \]
Property Tests

Int $\rightarrow p(x) \rightarrow ?$
Property Tests

\( \text{Int} \rightarrow p(x) \rightarrow ? \)

Invariant
Invariant: “a function, quantity, or property that remains unchanged when a specified transformation is applied.”
Basic Property Tests
What is true about addition?
\[ x + 0 = x \]
test "addition with zero returns the same number" do
end

def add(_x, _y) do
end
test "addition with zero returns the same number" do
  ptest do
    end
  end
end

def add(_x, _y) do
  end
test "addition with zero returns the same number" do
  ptest x: int() do
    end
end

def add(_x, _y) do
end
test "addition with zero returns the same number" do
  ptest x: int() do
    assert add(x, 0) == x
  end
end

def add(_x, _y) do
end
test "addition with zero returns the same number" do
  ptest x: int() do
    assert add(x, 0) == x
  end
end

def add(_x, _y) do
end
test "addition with zero returns the same number" do
  ptest x: int() do
    assert add(x, 0) == x
  end
end

def add(x, _y) do
  x
end
x + y == y + x
test "addition is commutative" do
end

def add(x, _y) do
  x
end
test "addition is commutative" do
  ptest x: int(), y: int() do
  x
  end
end

def add(x, _y) do
  x
  end
test "addition is commutative" do
  ptest x: int(), y: int() do
    assert add(x, y) == add(y, x)
  end
end

def add(x, _y) do
  x
end
keathley in ~/D/qc_example on master
> test
test "addition is commutative" do
  ptest x: int(), y: int() do
    assert add(x, y) == add(y, x)
  end
end

def add(x, _y) do
  x
end
test "addition is commutative" do
  ptest x: int(), y: int() do
    assert add(x, y) == add(y, x)
  end
end

def add(x, y) do
  x * y
end
test "addition is commutative" do
  ptest x: int(), y: int() do
    assert add(x, y) == add(y, x)
  end
end

def add(x, y) do
  x * y
end
test "addition is commutative" do
  ptest x: int(), y: int() do
    assert add(x, y) == add(y, x)
  end
end

def add(x, 0), do: x

def add(x, y) do
  x * y
end
\[(1 + x) + y = x + (1 + y)\]
def add(x, y)
do
  x * y
end

def add(x, 0), do: x
def add(x, y) do
  x * y
end

test "addition is associative" do
end
test "addition is associative" do
  ptest x: int(), y: int(), z: int() do
    end
end

def add(x, 0), do: x

def add(x, y) do
  x * y
end
test "addition is associative" do
  ptest x: int(), y: int(), z: int() do
    assert add(x, add(y, z)) == add(add(x, y), z)
  end
end

def add(x, θ), do: x

def add(x, y) do
  x * y
end
test "addition is associative" do
  ptest x: int(), y: int(), z: int() do
    assert add(x, add(y, z)) == add(add(x, y), z)
  end
end

def add(x, 0), do: x
def add(x, y) do
  x * y
end
keathley in ~/D/qc_example on master
λ # test
def add(x, 0), do: x
def add(x, y) do
  x * y
end
def add(x, y) do
  x + y
end
A Real-ish Example
Modeling the application

\[ p(x) \]
Modeling the application

What is the input domain?
Vote
Vote
Vote
User
Modeling Users as FSMs

- **logged_out**
- **logged_in**
- **login**
- **logout**
- **vote**
Generate Commands
Generate Commands
Generate Commands
Generate Commands
Generate Commands
Generate Commands
[{:vote, "chris", 1},
  {:vote, "chris", 2},
  {:vote, "jane", 1},
  {:vote, "jane", 1},
  {:vote, "jane", 3},
  {:vote, "chris", 2}]

Generated Commands

```
[{
  :vote, "chris", 1,
},
{
  :vote, "chris", 2,
},
{
  :vote, "jane", 1,
},
{
  :vote, "jane", 1,
},
{
  :vote, "jane", 3
},
{
  :vote, "chris", 2
}]
```

```
[{
  :vote, "jane", 1
}]
```
Property: Users votes should increase
Property: Users votes should increase

test "users votes increase after voting" do
end
Property: Users votes should increase

test "users votes increase after voting" do
test [commands: gen_commands("chris")]
done
done
Property: Users votes should increase

test "users votes increase after voting" do
  ptest [commands: gen_commands("chris")]
    do
      VoteCounter.reset()
    end
  end
end
Property: Users votes should increase

test "users votes increase after voting" do
  ptest [commands: gen_commands("chris")]
    do
      VoteCounter.reset()
      {_state, result} = run_commands(commands, Client)
    end
  end
end
test "users votes increase after voting" do
  ptest [commands: gen_commands("chris")] do
    VoteCounter.reset()
    {_state, result} = run_commands(commands, Client)
    assert result
  end
end
Property: Users votes should increase

test "users votes increase after voting" do
  ptest [commands: gen_commands("chris")]
    do
      VoteCounter.reset()
     ({_state, result} = run_commands(commands, Client))
      assert result
    end
end

def run_commands(commands, module) do
  Enum.reduce
    (commands,
      {0, true},
      & run_command(module, &1, &2))
end
def gen_commands(name) do
end
def gen_commands(name) do
  list(of: gen_vote(name), max: 20)
end
def gen_commands(name) do
  list(of: gen_vote(name), max: 20)
end

def gen_vote(name) do
end
def gen_commands(name) do
    list(of: gen_vote(name), max: 20)
end

def gen_vote(name) do
    tuple(like: {
    })
end
Command Generators

def gen_commands(name) do
  list(of: gen_vote(name), max: 20)
end

def gen_vote(name) do
  tuple(like: {
    value(:vote),
  })
end
def gen_commands(name) do
  list(of: gen_vote(name), max: 20)
end

def gen_vote(name) do
  tuple(like: {
    value(:vote),
    value(name),
  })
end
Command Generators

```python
def gen_commands(name) do
  list(of: gen_vote(name), max: 20)
end

def gen_vote(name) do
  tuple(like: {
    value(:vote),
    value(name),
    choose(from: [value(1), value(2), value(3)])
  })
end
```
defmodule ClientStateMachine do
end
defmodule ClientStateMachine do
  def vote(name, id) do
    %{"votes" => new_votes} = post(id, name)
    {:ok, new_votes}
  end
end
defmodule ClientStateMachine do
  def vote(name, id) do
    %{"votes" => new_votes} = post(id, name)
    {:ok, new_votes}
  end

  def vote_next(state, [id, name], _result) do
    {:ok, update_in(state, [name, to_string(id)], &(&1 + 1))}
  end
end
defmodule ClientStateMachine do
  def vote(name, id) do
    %{"votes" => new_votes} = post(id, name)
    {:ok, new_votes}
  end

  def vote_next(state, [id, name], _result) do
    {:ok, update_in(state, [name, to_string(id)], &(&1 + 1))}
  end

  def vote_post(state, [id, name], actual_result) do
    expected_result = get_in(state, [name, to_string(id)]) + 1
    {:ok, actual_result == expected_result}
  end
end
Property: Users votes should increase

test "users votes increase after voting" do
  ptest [commands: gen_commands("chris")] do
    VoteCounter.reset()
    {_state, result} = run_commands(commands, Client)
    assert result
  end
end
keathley in ~/D/phoenix_qc_example on multiple-clients is unpushed
Property: Users shouldn’t effect other users votes
Property: Users should get the correct votes

test "users don't effect each others votes" do
end
Property: Users should get the correct votes

test "users don't effect each others votes" do
  ptest [chris: gen_commands("chris"), jane: gen_commands("jane")) do
  end
end
Property: Users should get the correct votes

test "users don't effect each others votes" do
  ptest [chris: gen_commands("chris"), jane: gen_commands("jane")) do
    VoteCounter.reset()
  end
end
test "users don't effect each others votes" do
do  ptest [chris: gen_commands("chris"), jane: gen_commands("jane")]
do    VoteCounter.reset()
        {_state, result} = run_commands([chris, jane], Client)
    end
end
end
Property: Users should get the correct votes

test "users don't effect each others votes" do
  ptest [chris: gen_commands("chris"), jane: gen_commands("jane")]
    do
      VoteCounter.reset()
      {_state, result} = run_commands([chris, jane], Client)
      assert result
    end
end
Property: Users should get the correct votes

test "users don't effect each others votes" do
  ptest [chris: gen_commands("chris"), jane: gen_commands("jane")]
    do
      VoteCounter.reset()
      {_state, result} = run_parallel_commands([chris, jane], Client)
      assert result
    end
end
Running parallel tests

def run_parallel_commands([l1, l2], module) do
    t1 = Task.async(fn -> run_commands(l1, module) end)
    t2 = Task.async(fn -> run_commands(l2, module) end)
    {_, ra} = Task.await(t1)
    {_, rb} = Task.await(t2)
    {:ok, ra && rb}
end
keathley in ~/D/phoenix_qc_example on multiple-clients is unpushed

mix test --only focus
def new(conn, %{"id" => id, "name" => name}) do
  { :ok, current_votes } = VoteCounter.get(id)
  new_votes = [name | current_votes]
  VoteCounter.put(id, new_votes)

  # Other nonsense
end
The Bug

Votes
The Bug

3 Votes
The Bug

3

3

Votes
The Bug

4

4

Votes
The Bug

4 Votes
The Bug

3 + 1 + 1 == 4

?
The Bug

def new(conn, %{"id" => id, "name" => name}) do
  {:ok, current_votes} = VoteCounter.get(id)
  new_votes = [name | current_votes]
  VoteCounter.put(id, new_votes)

  # Other nonsense
end
def new(conn, %{"id" => id, "name" => name}) do
  {:ok, new_votes} = VoteCounter.incr(id, name)

  # Other nonsense
end
keathley in ~/D/phoenix_qc_example on multiple-clients is unpushed
Conclusion
How to think in properties
Generating data
Generate commands
Model users as FSMs
Resources:

“QuickCheck: A lightweight tool for Random Testing of Haskell Programs”

“Finding Race conditions in Erlang with QuickCheck and PULSE”

Testing Async apis with QuickCheck
https://www.youtube.com/watch?v=iW2J7Of8jsE&t=272s

Composing Test Generators
https://www.youtube.com/watch?v=4-sPhFtGwZk

Property based testing for better code
https://www.youtube.com/watch?v=shngiiBfD8o
If you could write less code and find more bugs would you do that?
Thanks

Chris Keathley / @ChrisKeathley / c@keathley.io