Sustainable Testing

potatosalad

September 7, 2018

Sustainable Testing

gitpitch.com/potatosalad/elixirconf2018

Andrew Bennett

potatosalad

potatosaladx

TOYOTA
connected

Symptoms of Unsustainable Testing

- Small code change = every test case fails
- Bugs are rarely discovered before production
- Non-critical systems are over-tested
- Everyone is unhappy :-(

Implementation details
change quite often,
and the test suites
are to be long lived.

— *Common Test: Why Test?*

Why Test?

- Prove that a program is correct.

Proof of No Proof

- The Impossibility of Complete Testing
- Tests Cant Prove The Absence Of Bugs
- Proofs Cant Prove The Absence Of Bugs
- StackOverflow: Why can't programs be proven?
- Wikipedia: Formal verification

Why Test?

- ~~Prove that a program is correct.~~
- Find bugs.

A successful test suite is one that reveals a bug.

> *If a test suite results in OK,*
> *then we know very little*
> *that we did not know before.*
>
> — *Common Test: Why Test?*

Techniques

- TDD - Test Driven Development
- Acceptance Testing
- Code Coverage
- Model Checking / Property-based testing

Model Checking

- TLA+
- Dezyne
- Concuerror
- QuickCheck, PropEr, and StreamData

PropCheck

(which uses PropEr)

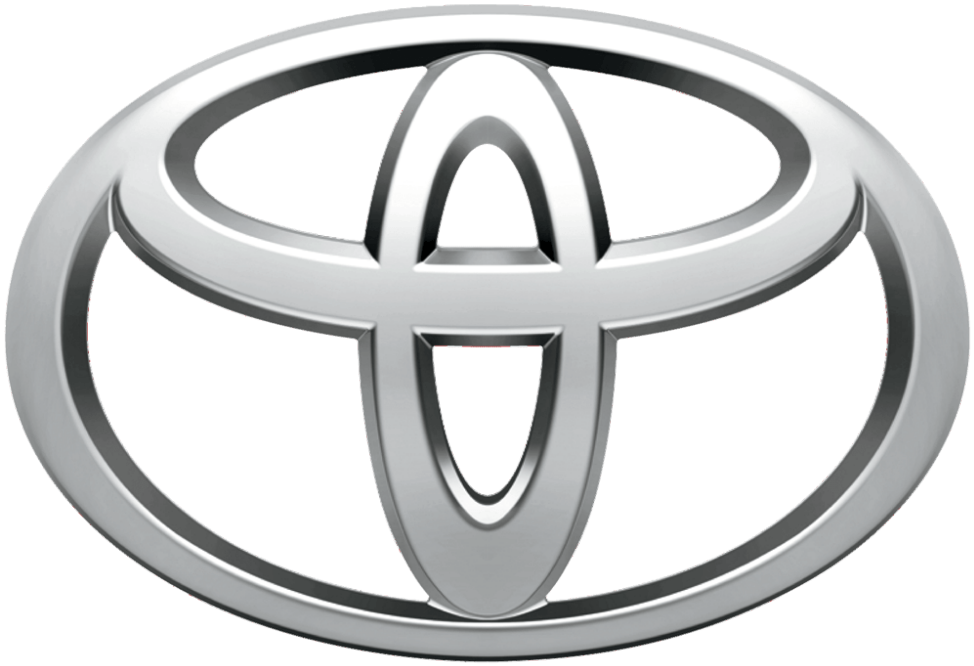
Toyota

propertesting.com

Terminology

Testing Levels

Level	Box	Automatability	Value
Unit	White	Highest	Lowest
Integration	Gray	High	Low
System	Black	Low	High
Operational	Black	Lowest	Highest



TOYOTA

自働化

Jidoka

Autonomation

"Automation with a Human Touch"

改善

Kaizen

Reform Good

"Continuous Improvement"

現地現物

Genchi Genbutsu

Actual Place　*Actual Thing*

“Go and See”

Toyota Production System

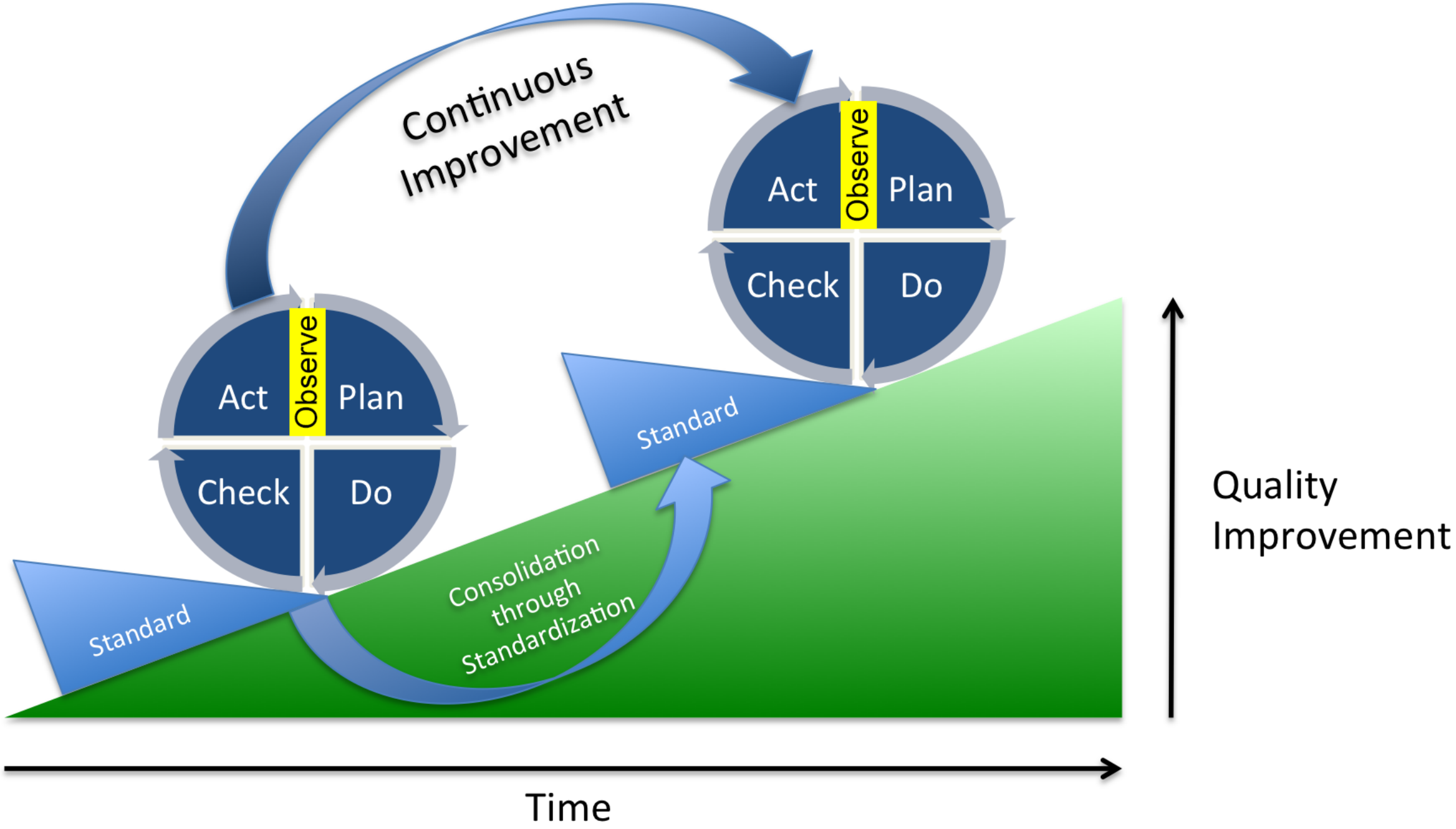
principles for

Sustainable Testing

1. Jidoka — “Automation with a Human Touch”
2. Kaizen — “Continuous Improvement”
3. Genchi Genbutsu — “Go and See”

OPDCA Cycle

- Observe
- Plan
- Do
- Check
- Act


Continuous Improvement
Act
Observe
Plan
Check
Do
Act
Observe
Plan
Check
Do
Standard
Standard
Consolidation through Standardization
Quality Improvement
Time

Why Test?

Find Bugs

Testing Levels

Level	Box	Automatability	Value
Unit	White	Highest	Lowest
Integration	Gray	High	Low
System	Black	Low	High
Operational	Black	Lowest	Highest

Unit Testing

```
01 defmodule Example do
02   def smallest([head | tail]) do
03     smallest(tail, head)
04   end
05
06   defp smallest([head | tail], min) when head < min,
07     do: smallest(tail, head)
08
09   defp smallest([_head | tail], min),
10     do: smallest(tail, min)
11
12   defp smallest([], min),
13     do: min
14 end
```

```elixir
Example.smallest([1, 2]) == 1
Example.smallest([:b, :a]) == :a
Example.smallest([:a, 0]) == 0
```

```elixir
defmodule ExampleTest do
  use ExUnit.Case

  test "finds smallest element" do
    assert Example.smallest([1, 2]) == 1
    assert Example.smallest([:b, :a]) == :a
    assert Example.smallest([:a, 0]) == 0
  end
end
```

```elixir
def model_smallest(list) do
  List.first(Enum.sort(list))
end
```

```elixir
01 defmodule ExampleTest do
02   use ExUnit.Case
03
04   test "finds smallest element" do
05     assert Example.smallest([1, 2]) == model_smallest([1, 2])
06     assert Example.smallest([:b, :a]) == model_smallest([:b, :a])
07     assert Example.smallest([:a, 0]) == model_smallest([:a, 0])
08   end
09
10   def model_smallest(list) do
11     List.first(Enum.sort(list))
12   end
13 end
```

```elixir
defmodule ExampleTest do
  use ExUnit.Case
  use PropCheck

  property "finds smallest element" do
    forall list <- non_empty(list(term())) do
      Example.smallest(list) == model_smallest(list)
    end
  end

  def model_smallest(list) do
    List.first(Enum.sort(list))
  end
end
```

Integration Testing

The Pensieve is enchanted to recreate memories…so that either the owner, or (and herein lies the danger) a second party, is able to enter the memories…

— *"Pottermore: Pensieve" by J.K. Rowling*

```
01  pensieve = Pensieve.start()
02  memory_ref = Pensieve.store(pensieve, "my memory")
03
04  :ok = Pensieve.load(pensieve, memory_ref)
05  {:ok, "my memory"} = Pensieve.relive(pensieve)
06
07  :ok = Pensieve.stop(pensieve)
```

Pensieve State Machine

```
01  :stopped -> [:empty]
02
03  :empty -> [{:loaded, memory}, :stopped]
04
05  {:loaded, memory} -> [:empty, :stopped]
```

plus the `data` which stores the memories

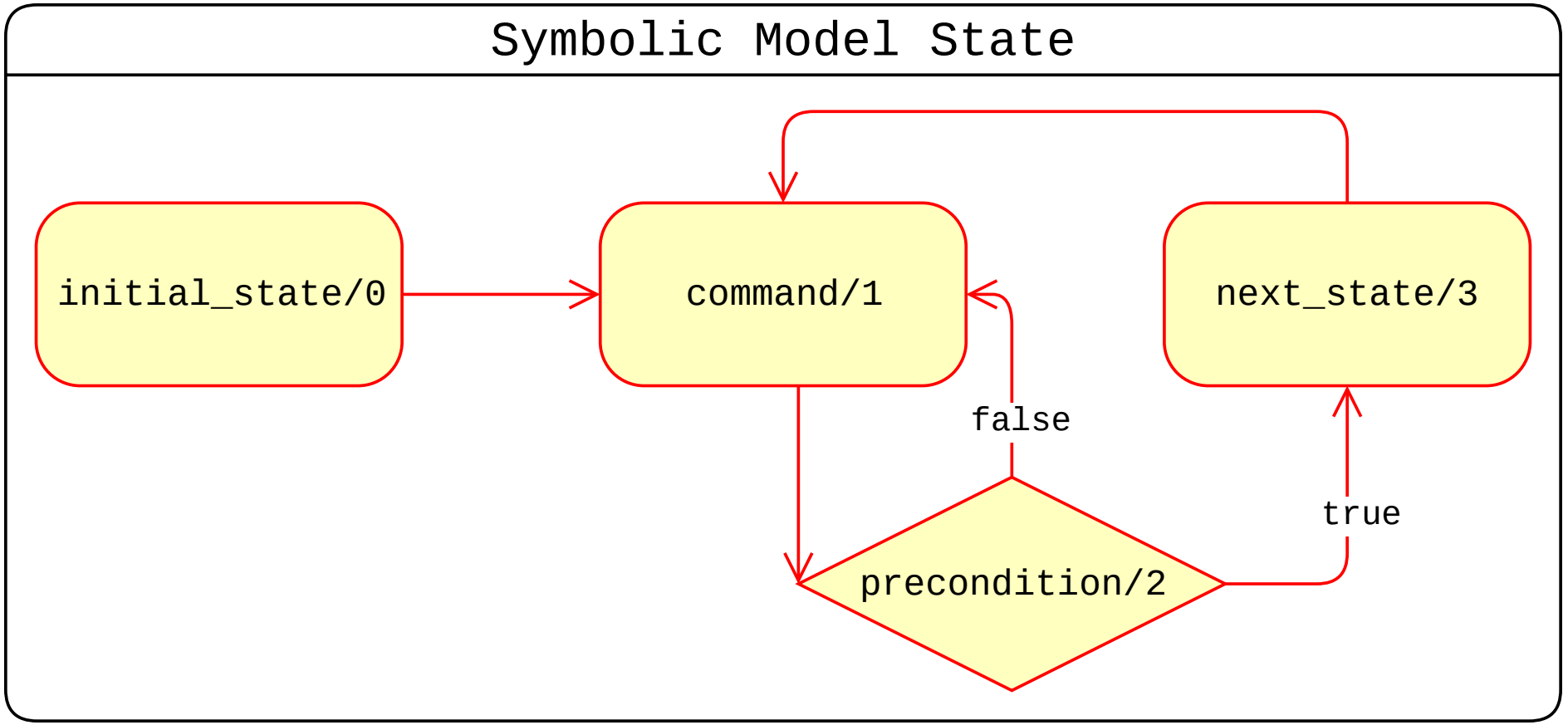
```elixir
defmodule PensieveModel do
  defstruct state: :stopped, memories: %{}

  def new() do
    %__MODULE__{}
  end

  def start(p = %__MODULE__{state: :stopped}) do
    p = %{p | state: :empty, memories: %{}}
    {:ok, p}
  end

  def stop(p = %__MODULE__{state: state}) when state != :stopped do
    p = %{p | state: :stopped, memories: %{}}
    {:ok, p}
  end

```

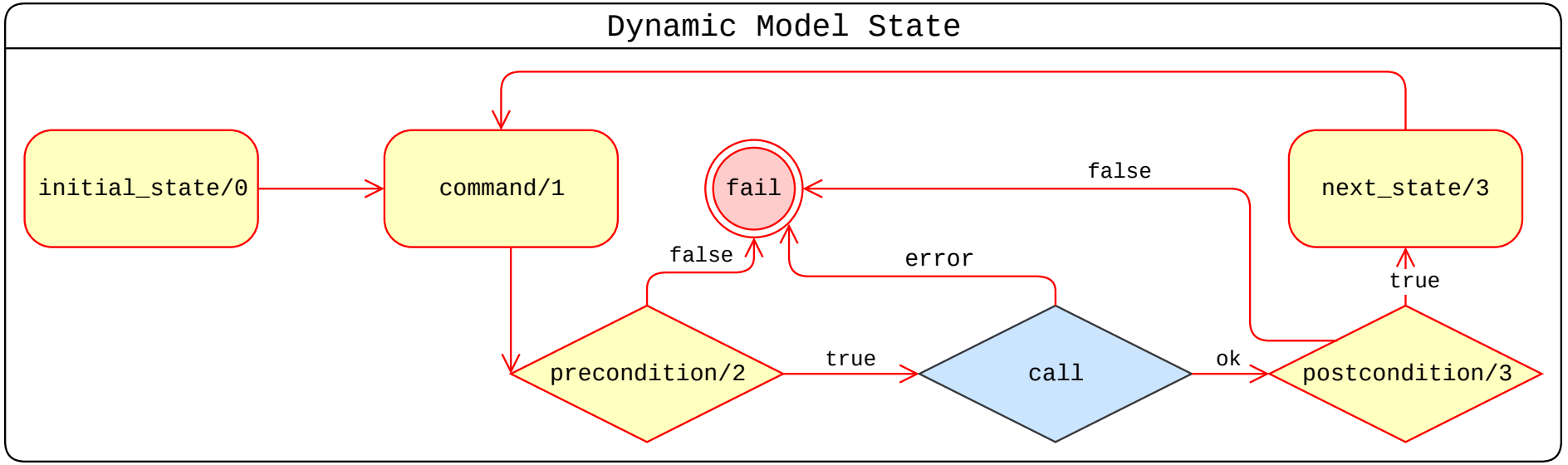
```
01  test "pensieve model static" do
02    # setup
03    {:ok, apps} = Application.ensure_all_started(:pensieve)
04    model = PensieveModel.new()
05    # start
06    system = Pensieve.start()
07    assert(is_pid(system))
08    assert({:ok, model} = PensieveModel.start(model))
09    # store
10    value = :memory
11    key = Pensieve.store(system, value)
12    assert(is_reference(key))
13    assert({:ok, model} = PensieveModel.store(model, key, value))
14    # load
15    assert(:ok = Pensieve.load(system, key))
16    assert({:ok, model} = PensieveModel.load(model, key))
17    # relive
```

System Testing

`PropCheck.StateM`

(or `:proper_statem`)


Symbolic Model State
initial_state/0
command/1
next_state/3
false
true
precondition/2


Dynamic Model State
initial_state/0
command/1
fail
false
next_state/3
false
error
true
precondition/2
true
call
ok
postcondition/3

```
01 # Failing command sequence:
02 [
03   {:set, {:var, 1}, {:call, Pensieve, :start, []}},
04   {:set, {:var, 2}, {:call, Pensieve, :store, [{:var, 1}, :x]}},
05   {:set, {:var, 3}, {:call, Pensieve, :load, [{:var, 1}, {:var, 2}]
06   {:set, {:var, 4}, {:call, Pensieve, :relive, [{:var, 1}]}}
07 ]
08 # At state:
09 {:p,
10  %PensieveModel{
11    memories: %{:r => :x},
12    state: {:loaded, :x}
13  }}
14 # Result:
15 {:postcondition, false}
16 # History:
17 [
```

```elixir
defmodule PensieveModelStateM do
  use PropCheck
  use PropCheck.StateM

  alias PensieveModel, as: M

  @impl :proper_statem
  def initial_state() do
    # structure: {system, model}
    {nil, M.new()}
  end

  @impl :proper_statem
  def command({_system, %M{state: :stopped}}) do
    oneof([
      {:call, Pensieve, :start, []}
    ])
```

```
01  defmodule PensieveModelTest do
02    use ExUnit.Case, async: false
03    use PropCheck
04    import PropCheck.StateM
05
06    alias PensieveModelStateM, as: M
07
08    property "pensieve model operations", [:verbose, {:numtests, 100}
09      forall(cmds <- commands(M)) do
10        {:ok, apps} = Application.ensure_all_started(:pensieve)
11        {history, state, result} = run_commands(M, cmds)
12        for app <- apps, do: Application.stop(app)
13
14        (result == :ok)
15        |> aggregate(command_names(cmds))
16        |> when_fail(print_failure_report(cmds, state, result, histor
17      end
```

```
01 ..................................................................
01 OK: Passed 100 test(s).
01
01 32% {'Elixir.Pensieve',start,0}
01 28% {'Elixir.Pensieve',store,2}
01 26% {'Elixir.Pensieve',stop,1}
01 8% {'Elixir.Pensieve',load,2}
01 3% {'Elixir.Pensieve',relive,1}
```

```
01 ..................................................................
01 OK: Passed 100 test(s).
01
01 15% {empty,{'Elixir.PensieveShim',load_known,2}}
01 11% {stopped,{'Elixir.PensieveShim',start,0}}
01 11% {loaded,{'Elixir.PensieveShim',load_known,2}}
01 11% {loaded,{'Elixir.PensieveShim',load_unknown,2}}
01 10% {loaded,{'Elixir.PensieveShim',store,2}}
01 10% {empty,{'Elixir.PensieveShim',relive,1}}
01 9% {loaded,{'Elixir.PensieveShim',relive,1}}
01 9% {empty,{'Elixir.PensieveShim',store,2}}
01 7% {empty,{'Elixir.PensieveShim',load_unknown,2}}
01 1% {empty,{'Elixir.PensieveShim',stop,1}}
01 1% {loaded,{'Elixir.PensieveShim',stop,1}}
```

`:proper_types.frequency/1`

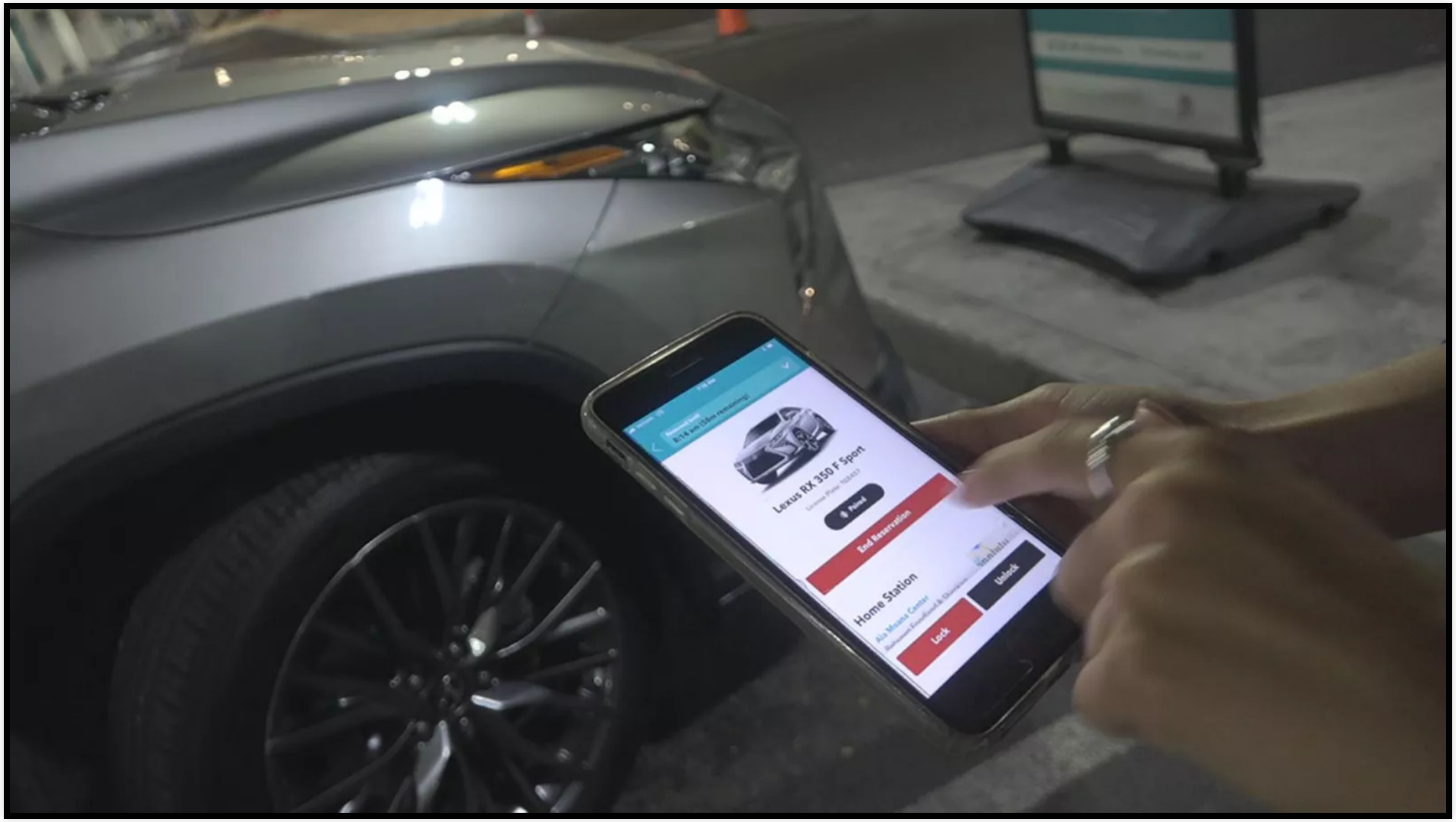
or

`:proper_types.weighted_union/1`

```
01  ==> share
01  ..................................................................
01  OK: Passed 100 test(s).
01  36% {'Elixir.ShareShim',add_driver_new,2}
01  29% {'Elixir.ShareShim',add_vehicle_new,2}
01  10% {'Elixir.ShareShim',set_time,1}
01  4% {'Elixir.ShareShim',start_reservation_valid,3}
01  4% {'Elixir.ShareShim',suspend_driver,2}
01  3% {'Elixir.ShareShim',approve_driver,2}
01  2% {'Elixir.ShareShim',start_reservation_unapproved_driver,3}
01  2% {'Elixir.ShareShim',stop_reservation_started,2}
01  1% {'Elixir.ShareShim',start_reservation_invalid_driver,3}
01  1% {'Elixir.ShareShim',start_reservation_invalid,3}
01  1% {'Elixir.ShareShim',stop_reservation_stopped,2}
01  1% {'Elixir.ShareShim',start_reservation_invalid_vehicle,3}
```

potatosalad/elixirconf2018

Operational Testing



Credit to KITV 4: Island News


RESERVED
24/7
HUI
SERVCO


Elevators
RESERVED
FOR CAR SHARING
HUI
SERVCO
@DriveHui
DriveHui.com
HUI MOBILITY PARTNER


IUH

Aim for finding bugs.

Write whatever test that has
the highest probability of finding a bug,
now or in the future.
Concentrate more on the critical parts.

— *Common Test: Why Test?*

potatosalad/elixirconf2018