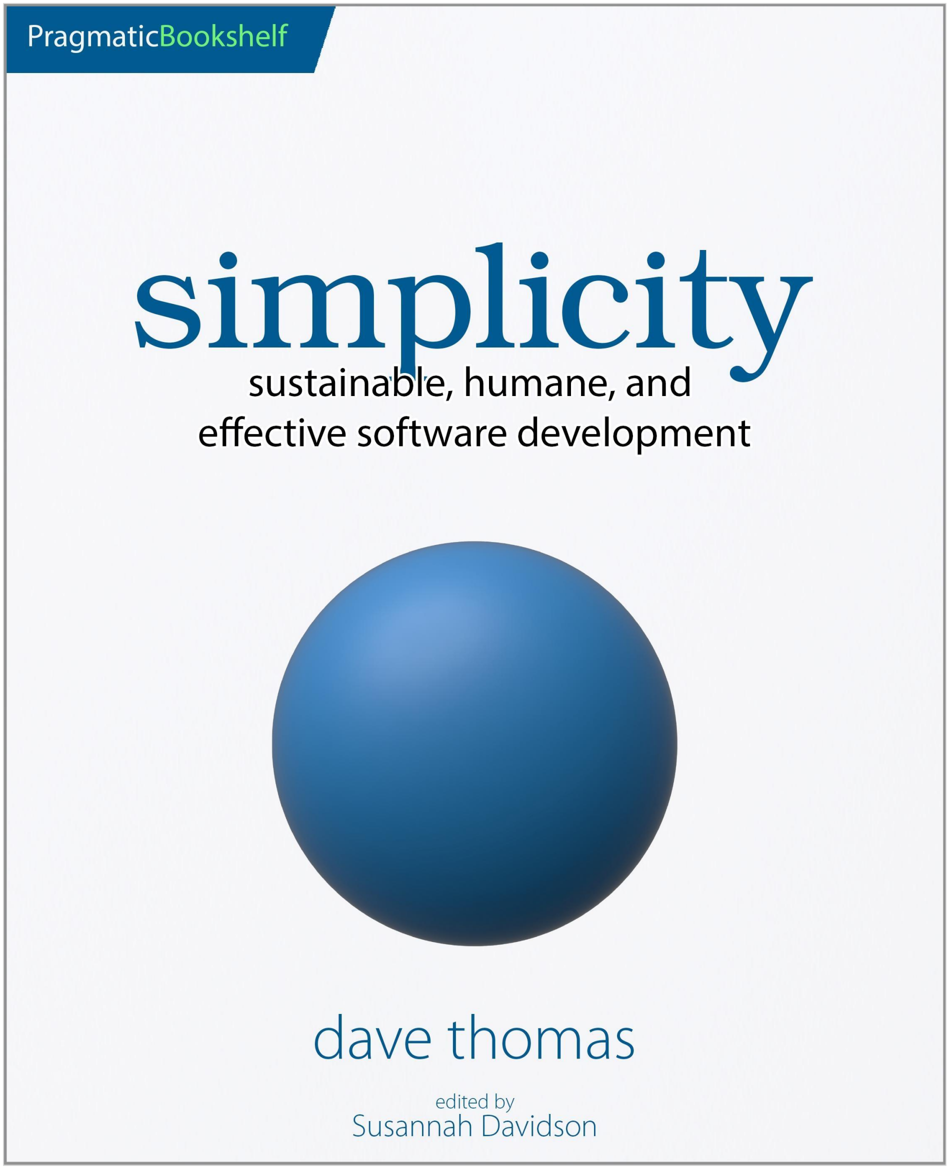


PragmaticBookshelf

simplicity

sustainable, humane, and effective software development

dave thomas

edited by
Susannah Davidson

Simplify Logic With State Machines

Table-driven code is great for simplifying repetitive sequences of code. State machines take this to the next level by adding context and the ability to decide dynamically which code to run in a given situation,

(You may see state machines referred to as *Finite State Automata* (FSA) or *Finite State Machines* (FSM).)

However, there's a lot of FUD spread directed at state machines. As a result many developers avoid them. That's just plain silly—a simple state machine can make your code far easier to understand and change.

Idea 63 **You don't need libraries or design patterns to write a state machine**

I use state machines all the time, and they have only ever improved the code I was working on. Here are a few examples:

- Parsing

  I needed to parse a file that was *almost* in comma separated variable (CSV), but it used backslash to escape characters.

- Extracting information from a Shopify Order

  Shopify orders are complex beasts, with many record type and intricate rules. A state machine helped me make sure the correct information was presented in the right order.

- Handling order fulfillment

  A simple state machine handled the changes in an order's fulfillment status, generating shipments, refunds, and notification as needed.

- Detecting patterns in log messages

  I had to analyze web server log messages in real time, looking for times where a particular URL was accessed three times within five seconds from the same IP address. Many such patterns could overlap in the stream of messages. This turned out to be really simple using a set of dynamically generated state machines.

The list goes on.

Whenever you are handling a sequence of *events*, where each event affects how the next event should be handled, consider using a state machine. If these events are spread out over time, with potentially long delays between each, the case for using a state machine is even more compelling. They'll make your life simpler.

Implementing State Machines

There's no need for complex libraries or long-winded pattern-based approaches when you need a state machine. All you need is a *lookup table*.

Let's start with a really simple state machine—a push-button pen. Each time you press the button, the tip extends if it was retracted and retracts if it was extended.

A state machine is defined by *events*, *states*, and *transitions*. The machine sits in a particular state until an event comes along. This may cause a transition into a different state.

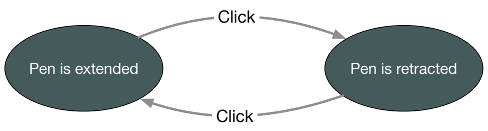
In the case of our retractable pen, we have:

- One event type: the button was pressed.
- Two states: extended and retracted.
- Two transitions: extended→retracted and retracted→extended.

If you click the top when the pen is extended, it transitions to being retracted, and if you click it when it's retracted, it transitions to extended.

We draw state machines with the states in boxes or circles, and the transitions as arrows between states. The transitions are labelled with the event that triggers them.

Here's a state machine diagram that describes our retractable ball-point pen.



This is a really simple diagram, but when they get more complicated, I like to see how they work by putting my finger on a state and then imagining an incoming event. I find the arrow labelled by that event, and my finger follows it to the next state.

What about the implementation? Well, this particular state machine is so simple that all we need is a boolean:

```
let penRetracted = true

eventStream.on("click", () => {
  penRetracted = !penRetracted
}
```

It may not look like much, but that's a working state machine. Still, let's get a little more complicated.

A More Complicated Example: A Keypad Lock

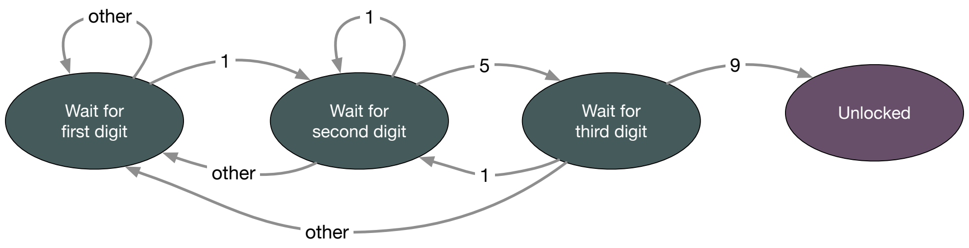
Here's a keypad lock. It opens when you press 1 → 5 → 9 in sequence.

- The user can press any number of keys before entering the correct sequence.
- If they enter a wrong digit, they must start over.
- If they enter 1 in the middle of a valid sequence, it should reset the sequence.

Try It

Before looking at the state diagram below, try sketching your own version. Use the finger technique do make sure it correctly unlocks given the sequences 159, 432159, 15159.

Here's my version of the state diagram:



The trick is returning to the "wait for second digit" state if a 1 is pressed in the middle of what started as a valid sequence.

Implementing the Keypad Lock

To implement this, I'd use a lookup table (a dictionary, hash, map, or object, depending on your language).

fsm/159.rb

```
TRANSITIONS = {
  "wait_for_first_digit" => {
    "1"     => "wait_for_second_digit",
    "other" => "wait_for_first_digit",
  },
  "wait_for_second_digit" => {
    "5"     => "wait_for_third_digit",
    "1"     => "wait_for_second_digit",
    "other" => "wait_for_first_digit",

  },
  "wait_for_third_digit" => {
    "9"     => "unlocked",
    "1"     => "wait_for_second_digit",
    "other" => "wait_for_first_digit",
  },
  "unlocked" => NIL
}
```

The entire state transition logic is encapsulated in this data structure.

- Each state is a key in the dictionary.
- The values are nested maps that define what state to transition to based on incoming input.

The code that handles this is trivial

fsm/159.rb

```
state = "wait_for_first_digit"
while TRANSITIONS[state] && key = gets()
  key   = key.strip
➤ state = TRANSITIONS[state][key] || TRANSITIONS[state]["other"]
end
```

In fact, the actual state machine implementation is just one line of code (it has the arrow in the margin).

But Wait, There's More

At this point I could dive deep into all the other cool things you can do with state machines. You can encode actions to take on each transition into the table. You can have long running state machines, which persist their state between events. You can use state machines to drive workflows. And you can turn much of the stuff that happens inside your code into events, meaning you can use state machines to implement the logic of handling them.

But that’s way too much detail for this book. If you’re interested (of course you are) I have an article about it.[1]

1. https://open.substack.com/pub/pragdave/p/simplify-logic-with-state-machines