Editors: Andy Hunt and Dave Thomas ■ The Pragmatic Programmers
andy@pra gmaticprogrammer.com ■ dave@pragmaticprogrammer.com

State Machines

Dave Thomas and Andy Hunt

We are surrounded by real-world state machines: ballpoint pen retractor mechanisms, vending machines, washing-machine controllers, digital watches. They are a trivial but underused technology that can simplify how we implement programs that must track how they got to their current state before handling a new event. However, many programmers feel that state machines are only useful when they're developing communication protocol stacks, which is not an everyday activity.

This is unfortunate. State machines can be appropriate in surprising circumstances. Correctly applied, they will result in faster, more modular, less coupled, and easier to maintain code. State machines make it easy to eliminate duplication, honoring the DRY principle.[1] They also let you write more expressive code, because you can specify intent and implementation independently. These are all good, pragmatic, reasons to investigate them further, so let's look at some simple state machine implementations and problems they can solve.

Stating the obvious

A state machine is a system with a set of unique states. One state is special—it represents the system's initial state. One or more of the other states are final states; when an event causes us to reach one of these the state machine exits. States are connected by transitions. Each transition is labeled with the name of an input event. When that event occurs, we follow the corresponding transition from the current state to arrive at the new state. State machines are often represented as diagrams with the states shown as circles and the transitions as labeled arrows between the states. Figure 1 shows a simple state machine that exits when a set of coin tosses results in a head, then a tail, and then another head. It starts in state S_0. If we toss a tail, the transition loops back and we stay in S_0; otherwise, we move on to S_1. This state moves to S_2 if we toss a tail next. From there we move on the S_3, a final state, if we see another head. This type of state machine is sometimes called a deterministic finite state machine or automaton. The graph in Figure 1 is a state transition diagram.

Using state machines

A state machine is useful whenever we have a program that handles input events and has multiple states depending on those events. These situations arise frequently in communications, parsing, emulations, and handling user input. You can spot a program that's a candidate for a state machine by looking for code that contains either deeply nested `if` statements or many flag variables. You can eliminate the flags and flatten the nesting using a state machine.

 0740-7459/02/$17.00 © 2002 IEEE

A while ago, Dave wrote a simple Web-based order-handling system. Customers could pay by check or purchase order. When orders were initially entered, a confirmation was mailed. When payment was received, the products were shipped. If that payment was a purchase order, the program generated an invoice and tracked its subsequent payment status. Because events could occur weeks apart, the status had to be tracked in a database.

After a while, the code started to get messy and handling the special cases began to get ugly. So, Dave reimplemented the code as a simple state machine. This state machine ended up having a dozen or so states and perhaps 15 action routines to deal with transitions between these states. The resulting code was a lot clearer (and a lot shorter). And when the customer changed the application's business rules, typically Dave just changed a few entries in the table that defined the state transitions.

However, that's a fairly complex example. Let's look at something simpler, such as a program that counts words in text. Here the input events are the characters we read, and the states are "W: in a word" and "S: not in a word." We can increment the word count whenever we transition from *S* to *W*. Figure 2 shows the state transition diagram. Note that we've added a semantic action to one of the transitions—we increment a count on the $S \rightarrow W$ transition. On its own, this example might not be particularly compelling—the code required to implement the state machine word counter is probably about the same size as the conventional version. However, say our requirement changed slightly—our client tells us that the program should now handle HTML files, ignoring any text between "<" and ">". We also deal with quoted strings, so that "Now is the <IMG src="clock.gif" alt="<time>"> for all good people" should count seven words, and correctly ignore the ">" in the quoted string. If we had taken the conventional approach, we'd now need flags for "skipping a command" and "in a quoted string." With the state machine, it is a simple extension, shown in Figure 3.

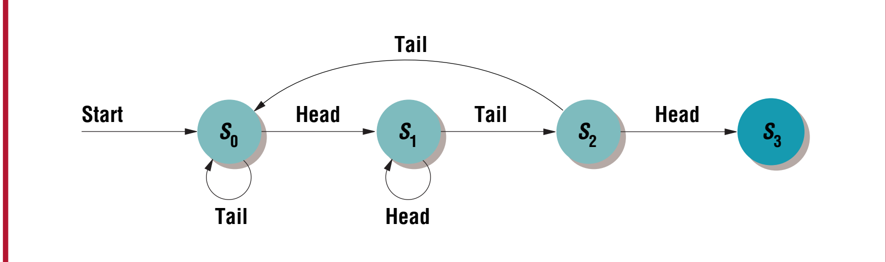

Figure 1. A state machines that exits for a head-tail-head sequence of coin tosses.

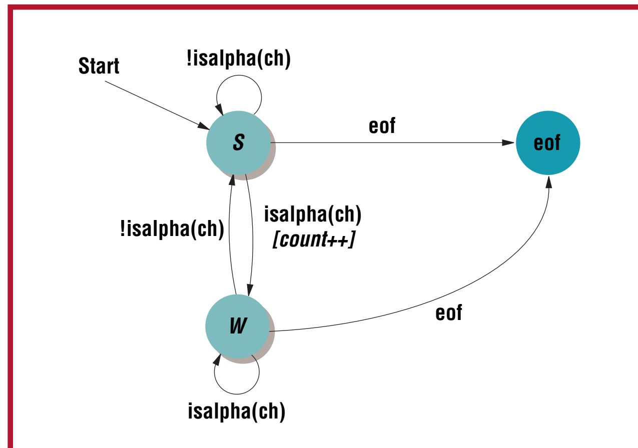

Figure 2. A state transition diagram that counts words in our text input.

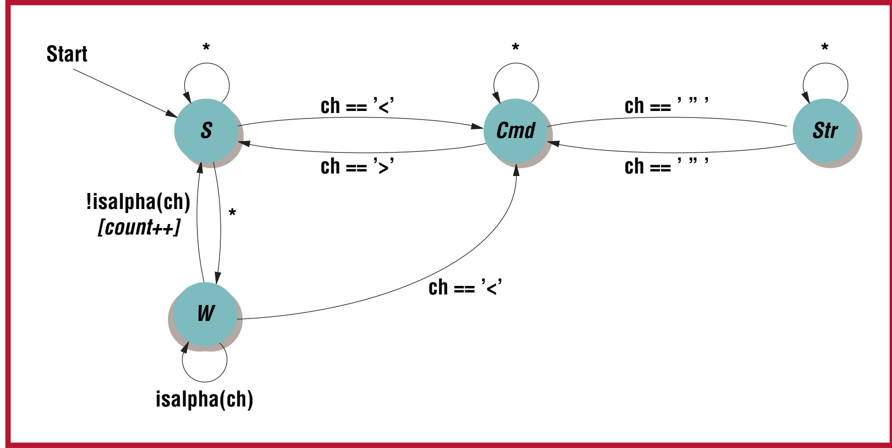

Figure 3. A state machine that counts words in HTML. We omitted the *eof* transitions (shown in Figure 2) for clarity.

Implementing state machines

For very simple state machines, we find it is easiest to implement the states and transitions manually. We use a variable to keep the current state and update it as events happen. Typically we'll have a `case` statement to handle the different states or events.

However, once we start becoming more complex, we convert the state transition diagram to a 2D table. The table is indexed by the current state and the input event, returning the resulting next state. It's convenient to include an action code in each table entry too, because this tells us what to do on each transition. These table

IEEE Software

How to Reach Us

Writers
For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access http://computer.org/software/author.htm.

Letters to the Editor
Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web
Access http://computer.org/software for information about *IEEE Software*.

Subscribe
Visit http://computer.org/subscribe.

Subscription Change of Address
Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address
Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles
For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.

entries are conveniently represented as structures or simple data-only classes. A more sophisticated implementation could replace these table entries with objects that include the behavior to be performed. Applications coded this way often have a trivial main loop:

```
while eventPending(){
 event = getNextEvent();
 entry = transitions
   [currentState][event];
 entry.executeAction();
 currentState =
   entry.nextState();
}
```

Once we have a program in this form, we can easily change it as new requirements come along. For example, if our client suddenly wants us to count HTML commands, we merely add a new action to the $S \rightarrow$ Cmd and $W \rightarrow$ Cmd transitions in the table. If we notice that we're handling HTML comments incorrectly, we just add a couple of new states and update the table accordingly—the main program doesn't change at all.

More sophisticated implementations

Once you get into the realm of large state machines, maintaining the state table manually becomes an error-prone chore. Rather than coding the table directly in our implementation language, we normally write a plain text file containing a simpler representation and use this to generate code. For the HTML word counter, our state file might start something like:

```
S: LT → (CMD NONE),
   WORD → (W INC),
   default → (S NONE)
W: LT → (CMD NONE),
   W → (W NONE),
   default → (S NONE)
```

Depending on the target language, we might then generate from this a header file containing the definitions of the states, events and actions, and a source file containing the transition table. The actions could be defined as enumerations or possibly as a set of function pointers. Robert Martin of Object Mentor implemented state machine compilers for Java and C++ based on these principles. You can download them from www.objectmentor.com/resources/downloads/index.

State machines and object-oriented development

If you're working in an object-oriented environment, the same basic principles apply. However, you can also use classes to provide a clean interface to the thing being modeled. In *Design Patterns*,[2] the Gang of Four present the State pattern. Their example is a TCP connection. As the connection changes state (presumably driven by an internal state transition system similar to the ones we discussed earlier), the connection object changes its behavior. When the connection is in the closed state, for example, a call to open it might succeed. However, if the connection is open already, the same call will be rejected. This is a tidy approach to managing the external interface to a state driven system.

State machines are an underused tool. The next time you find yourself adding "just one more flag" to a complex program, take a step back and see if perhaps a state machine might handle the job better.

References

1. A. Hunt and D. Thomas, "Don't Repeat Yourself," *The Pragmatic Programmer*, Addison-Wesley, Boston, 2000.
2. E. Gamma et al., *Design Patterns*, Addison-Wesley, Boston, 1995.

Dave Thomas and **Andy Hunt** are partners in The Pragmatic Programmers, LLC. They feel that software consultants who can't program shouldn't be consulting, so they keep current by developing complex software systems for their clients. Contact them via www.pragmaticprogrammer.com.