Finite state machines have long been used as an organizing principle for designing and implementing complex behavior in event-driven programs, such as network adapters and compilers. Now, programmable Web browsers have opened a new event-driven environment to a new generation of applications. Browser-based applications, popularized by Ajax, are becoming more complex. Designers and implementers can benefit from the discipline and structure that finite state machines offer. In this article, you learn how to use a finite state machine to design complex behavior for a simple Web widget -- an animated tooltip that fades into and out of view.

Part 2 in this series will describe how to implement that design in JavaScript, and take full advantage of its distinctive language features, including associative arrays and function closures. Part 3 will cover the practical issues of making the implementation work in all popular Web browsers. The resulting code is compact and concise, its logic is transparent, and its animation performs smoothly even on heavily loaded processors.

This article assumes you have some familiarity with JavaScript, and are interested in a deeper understanding of its more advanced features.

For years, Web designers have quietly exploited the JavaScript interpreters in popular Web browsers to enhance the appearance of their Web sites. They do this mainly by copying short snippets of code into their HTML pages. Now, with the recent popularity of Ajax, software engineers also use JavaScript to develop a new generation of applications that execute within browsers. As browser-based applications grow in size, they will increasingly require the same design patterns and development discipline that have evolved for other execution environments.

Browser-based applications execute in a real-time environment where mouse, keyboard, timer, network, and program events can occur at any time. When the behavior of an event-driven application depends upon the order in which events occur, its programming can become very
tangled, and consequently difficult to debug and modify. Software engineers have long used finite state machines, sometimes called discrete or deterministic finite automata in academic circles, as an organizing principle for developing event-driven programs.

The discipline imposed by finite state machines adds rigor to the design by replacing tangled logic with straightforward tables, resulting in simpler implementation and easier testing. Traditionally, finite state machines have proven helpful in developing programs as diverse as network drivers and compilers. They will be equally helpful in developing browser-based applications.

In this series, you'll develop a sample finite state machine application as an exercise to explore some distinctive features of the JavaScript language:

- Functions are first-class objects: they can be created, assigned to variables, and passed as arguments, just like any other object. Functions can be defined within another function and assigned to global variables or returned as results. Such functions will persist after the function that defines them has returned.
- Functions can refer to any variable in their lexical scope (the nested braces surrounding the function's definition), such as the local variables of a function that defines them. These variables become part of the function closure (the function, its own variables, and any variables it uses that are defined elsewhere in its lexical scope), and they also persist after the function that defines them has returned.
- Functions can be stored in associative arrays (arrays that are indexed by names rather than numbers).

These language features provide a compact and concise way to organize actions for events and transitions between states, and an elegant way to cope with differences between browser event models.

The sample application, FadingTooltip, is a more elaborate tooltip than the default built into most browsers. Tooltips created with the FadingTooltip widget use animation to fade into and out of view instead of popping up and abruptly vanishing, and they move with the cursor. The finite state machine pattern used to design this behavior makes its logic transparent. The JavaScript language features used to implement it make the source code compact and efficient.

This article shows how to design the behavior of an animated widget using the graph and table representations of a finite state machine. Subsequent articles will describe how to implement the table representation of a finite state machine in JavaScript, and how to handle the practical issues of testing the implementation in popular browsers.

**Basic tooltips**

Most modern graphical applications can temporarily display small text boxes containing helpful definitions, instructions, or suggestions when the cursor pauses over some visual control, such as a button, a selector, or an input field. These helpful text boxes were called "balloon help" in early Apple systems. They're called infopops in some IBM products, and ScreenTips in some Microsoft products. In this article, I use the more generic term tooltip.
Popular Web browsers such as Netscape Navigator, Microsoft Internet Explorer, Opera, and Mozilla Firefox will display tooltips for any HTML element that has a title attribute. For example, Listing 1 shows three HTML elements with title attributes.

**Listing 1. HTML code for browser tooltips**

```
Here are some <span title='Move your cursor a bit to the right, please.'> fields with built-in tooltips </span>: 
<input type='text' title='Type your bank account and PIN numbers here, please ...' size=25>
<input type='button' title='Go ahead. Press it. What's the harm? Trust me.' value='Press this button'>
```

The examples page shows how your browser renders HTML elements with the title attribute. Notice how the tooltips appear and disappear as you move your cursor over the elements. The text boxes contain simple text, without any formatting or styling. They pop into view after a short pause in cursor movement, and abruptly disappear after some arbitrary time elapses, or the cursor moves away from the HTML element, or a key is pressed. The browser never displays more than one text box at a time. The appearance and behavior of these tooltips are hard-wired into your browser; you cannot change them.

**More elaborate tooltips**

Built-in tooltips have much room for improvement, and recent versions of popular browsers provide all of the raw ingredients needed to build more elaborate tooltips. The HTML Division element creates a box that you can position anywhere in the browser's window. You can determine almost every aspect of the box's appearance with cascading style sheets (CSS). Cursor movements programmed in JavaScript can trigger particular actions for any visual element in the browser's window. You can program timers to sequence those actions.

If you use an older browser, you might not see some of this behavior. With the Opera browser prior to version 9, for example, the tooltips pop in and out, instead of fading in and out, because Opera was comparatively late in implementing the opacity style property. See Related topics to download current versions of popular browsers.

The examples page also contains some HTML elements with more elaborate tooltips. If you are running a recent version of a popular browser, you can compare the more elaborate tooltips to the built-in tooltips:

- They fade into view, and later fade from view, rather than popping in and out.
- They contain images as well as text, and are formatted and styled.
- While they are visible, they move with the cursor.
- The fading reverses direction as the cursor moves away from, and then returns to, an HTML element.
- More than one tooltip can be visible concurrently, some fading out while another fades in.

This enhanced behavior and appearance is not just cosmetic; it improves usability. Users who face busy pages with dozens or hundreds of elements might miss a tooltip that pops instantly into view. The human vision system, which is especially sensitive to motion, is more likely to notice a tooltip that fades into view and moves with the cursor, even when a distracted user's attention is...
focused elsewhere on the page. The images, formatting, and styling can convey information more effectively than unformatted text. And, all of the parameters of these more elaborate tooltips are configurable.

The rest of this article focuses on designing the FadingTooltip widget as a finite state machine. Subsequent articles will show how to implement and test the code. If you're in a hurry, though, Related topics links to the JavaScript source, and an HTML Web page that uses it, to produce the examples above.

**Finite state machines**

Finite state machines model behavior where responses to future events depend upon previous events. There is a rich body of academic literature in this field (see Related topics), but a useful working definition is straightforward. Finite state machines are computer programs that consist of:

- **Events** that the program responds to
- **States** where the program waits between events
- **Transitions** between states in response to events
- **Actions** taken during transitions
- **Variables** that hold values needed by actions between events

Finite state machines are most useful in situations where behavior is driven by many different types of events, and the response to a particular event depends on the sequence of previous events. The events that drive finite state machines can be external to the computer, originating from a keyboard, mouse, timer, or network activity, or they can be internal to the computer, originating from other parts of the application program, or other applications.

States are a way to remember previous events, and transitions are a way to organize responses to future events. One of the states must be designated as the initial state. There may be a final state, but this is optional, and the FadingTooltip widget does not have one.

Two common representations of finite state machines are:

**Directed graphs**

Balloons represent states, and arrows between them represent transitions, which are labeled with events and actions.

**Two-dimensional tables**

Rows and columns represent events and states, and cells contain actions and transitions. These representations are equivalent, but emphasize different aspects of design. Both are useful, and I use both later in this article.

Developing event-driven programs with finite state machines is a bit more complicated than ordinary procedural programming; it requires more discipline generally, and more design effort in particular. When done well, it can result in simpler code, faster testing, and easier maintenance. Even so, the complexity of finite state machines is not worthwhile for all event-driven programs. When the variety of events is small, for example, or the actions triggered by events are always the same, the extra development effort might not be justified.
Finite state machines and the runtime environment

Finite state machines are event-driven, and need a way to hook themselves to the events of interest in their runtime environment. These hooks, called event handlers, are very small fragments of code inserted into the runtime environment so they will execute whenever particular events occur. When they execute, event handlers need to obtain some basic information:

- The type of event that has occurred (for example, the cursor has moved, or a timer has expired)
- The context of the event (for example, which HTML element the cursor is over, or which network request has completed)
- The location of the finite state machine's own variables and methods

JavaScript is well suited for building event-driven finite state machines. Indeed, JavaScript is perhaps too well suited -- it has three different ways to hook events. Each of these event models is straightforward, but programs must implement all three to ensure that they will run on all popular browsers. The context of an event is passed directly to event handlers in two of these event models; for the other one, JavaScript function closures allow an event's context to be enclosed with its event handler.

JavaScript provides an object model that might seem a bit peculiar to Java and C++ programmers, but is nonetheless entirely adequate for coding the variables and methods of finite state machines. And, JavaScript associative arrays allow you to code the two-dimensional table representation of a finite state machine directly.

Methodically designing behavior

The fundamental ingredients for a finite state machine are the events it responds to, and the states in which it waits between events. The design must consider each possible event for each possible state:

- Whether the event can possibly happen in that state
- What actions should be taken to handle the event
- Which state to transition to after the event
- What variables need to be remembered between events

I begin the design procedure with a graph, as seen in Figure 1, that shows states as balloons and transitions as arrows connecting the balloons. I end with a table, as shown in Figure 4, that lists events and states as row and column headings, respectively. Each cell in the table lists the actions to be performed when a particular event occurs in a particular state, or else indicates that the event cannot occur in that state.

You usually need several iterations of this design procedure to get the graph and table right. For finite state machines with many events and states, the procedure can be quite tedious, and some discipline is needed to work methodically through each cell in the table during each iteration. This forces you to consider what behavior you want in every possible situation. You might discover opportunities to further elaborate or refine the behavior, or discover you need more (or fewer)
states than initially thought, or you might have to shuffle actions between cells to specify the right behavior in every situation.

This methodical procedure for designing finite state machines, though tedious, is well worthwhile. The completed table, as shown in Figure 4, reveals all of the logic for the behavior, and can be translated directly into code (see the actionTransitionFunctions source code).

**About JavaScript capabilities**

To design the FadingTooltip widget, you need to know some of JavaScript's capabilities. In the spirit of top-down design, I'll sketch the basic ideas here, and defer the details of implementation to the next article in this series.

All popular browsers can pass events to JavaScript code when the cursor passes over any HTML element on the page. These events are called `mouseover`, `mousemove`, and `mouseout`, indicating that the cursor has moved onto, is moving around above, and has moved away from the HTML element. The browser passes the current cursor position with these events. When the events occur, JavaScript can be programmed to dynamically create HTML Division elements, fill them with text, images, and markup, and position them near the cursor.

Browsers do not have native fade-in or fade-out functions, but you can simulate this by varying the transparency (well, actually the opacity, which is the opposite of transparency) of a Division element over time.

JavaScript has two types of timers: one-shot timers that generate a timeout event when they expire, and repeating tickers that generate timetick events periodically. You will need both for the FadingTooltip widget.

**Sketching a state graph**

Begin your design by reviewing the basic behavior that you want of the FadingTooltip widget. When the cursor passes over a particular HTML element, you want the widget to wait for the cursor to pause over that element. If it does, then you want the widget to fade the tooltip into view, display it for a while, and then fade it away.

Your finite state machine will need to respond to these events:

- The browser can pass mouseover, mousemove, and mouseout events to JavaScript when the cursor passes over a particular HTML element, moves around within it, and eventually leaves the element, respectively.
- JavaScript can program timeout events to indicate when the cursor has paused long enough or the tooltip has displayed long enough, and timetick events to animate increases or decreases in the opacity of the tooltip that simulate fade-in or fade-out, respectively.

You'll invent some states for the machine to wait in between events. Let's call the widget's initial state Inactive, which is where it will wait to be activated by a mouseover event. The widget will wait in Pause state until a timeout event indicates that the cursor has paused long enough over the HTML element. Then the widget will wait in FadeIn state while animating the fade-in with timetick
events, and then wait in Display state for another timeout event. Eventually, the widget will wait in FadeOut state while animating the fade-out with more timetick events. The widget will return to Inactive state, where it will wait for another mouseover event.

Figure 1 sketches this progression as a graph, representing the states as balloons, the transitions as arrows connecting them, and the events as labels on the arrows. A double border indicates the balloon for the initial state.

**Figure 1. Initial sketch of state graph**

![State Graph](image)

The FadingTooltip widget will have to take some actions for each event it handles:

- When a mouseover event occurs in Inactive state, it will have to start a one-shot timer before waiting in Pause state.
- When that timeout event occurs, it will have to create the tooltip (with an initial opacity of zero) and start a repeating ticker before waiting in FadeIn state.
- Each time a timetick event occurs, it will have to increase the tooltip's opacity slightly. When the tooltip's maximum opacity is reached, it will have to cancel the ticker and start another timer before waiting in Display state.
- When that timeout event occurs, it will have to start another ticker before waiting in FadeOut state.
- Each time a timetick event occurs in FadeOut state, it will have to decrease the tooltip's opacity slightly. When the tooltip's opacity reaches zero, the widget will cancel the ticker, delete the tooltip, and return to the Inactive state, where it will wait to be activated again by another mouseover event.

Figure 2 records these actions in the sketch by listing them beneath the event that triggers them.

**Figure 2. Initial sketch of state graph with actions appended to events**

![State Graph with Actions](image)
Translating a state graph into a state table

The graph representations shown above are a good way to start the design of a finite state machine. But the table representation is better for completing the design because it exposes all of the combinations of events and states to consider.

To translate a state graph into a state table, label the rows with event names and the columns with state names. The order of the names is arbitrary; I put the initial state in the first column and the initiating event in the first row. Then, copy the actions and next state for each event into the appropriate cell of the table, as shown in Figure 3.

Figure 3. Initial state table corresponding to initial state graph

<table>
<thead>
<tr>
<th></th>
<th>Inactive</th>
<th>Pause</th>
<th>FadeIn</th>
<th>Deploy</th>
<th>FadeOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>move</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Completing the state table

To complete the design of your finite state machine, consider each empty cell in the table. You need to consider, for each cell, whether that event can occur in that state, and if so, what actions your widget should take in that situation, and what the next state should be. This is the tedious but very necessary part of the design process.

The order in which you consider the cells doesn't matter. It's common to iterate over this step during the design process many times, considering each cell repeatedly, revising their contents frequently, in different orders each time. It is also common to add (or remove) states as a design evolves, prompting further revisions. I'll skip the iterations and summarize this design step by reviewing the results for each state and event in turn.

Inactive state

During this state, only the initiating event that you already considered should occur, sincemousemove and mouseout events should be preceded by a mouseover event, and no timers are running. So, you can mark all other cells in the first column "should not occur."

Before moving on, though, consider the mouseover event for this state a bit further. When you eventually create an HTML Division element for the tooltip, you need to position it near the cursor, so you want to save the current cursor position, which the browser passes with this event. And, it's good practice to cancel any running timer before starting a new timer. Add these actions to the mouseover cell.

Pause state

While waiting for its timer to expire, the cursor might move within the HTML element or move away from the HTML element. Decide what actions to take if these events occur, and what
the next state should be. If a mouseout event occurs in this state, you want the FadingTooltip widget to return to Inactive state, as if the cursor had never passed over the HTML element at all, but you do need to cancel your timer. Record this action and transition in the mouseout cell.

On the other hand, for a mousemove event, you want the widget to continue waiting for the cursor to pause, which requires that you cancel and re-start the timer. Because you want the tooltip to appear near the cursor, you want to update your saved cursor position. On review, you might realize that the actions and transition for a mousemove event in Pause state are the same as for a mouseover event in Inactive state. Rather than repeat everything in both cells, indicate this duplication directly in the mousemove cell. Mark all other cells in this column "should not occur."

**FadeIn state**

During this state, while simulating fade-in with timetick events, the cursor can continue to move around. If a mousemove event occurs, move the tooltip to match, and remain in the current state. If a mouseout event occurs, transition to FadeOut state, with the ticker still running, so that subsequent timetick events will decrease the tooltip's opacity from its current value. Record these events and transitions in the appropriate cells, and mark all other cells in this column "should not occur."

**Display state**

The cursor can, of course, continue to move around. If it moves within the HTML element, you'll want to take the same action as for FadeIn state -- move the tooltip to match. If it moves away from the HTML element, take the same action and transition as for a timeout event in Display state. Indicate both of these duplications directly in the mousemove and mouseout cells, and mark all other cells as "should not occur."

**FadeOut state**

During this state, while simulating fade-out with timetick events, the cursor can still move around. If it moves within the HTML element, take the same action as for FadeIn and Display states. If the mouse moves away from the HTML element, you need not do anything -- the ticker will continue to run, so that subsequent timetick events will decrease the tooltip's opacity from its current value until it reaches zero. Don't mark this cell "should not occur," but do indicate that no actions are needed. And, if the cursor moves back over the HTML element, move the tooltip back to the cursor and return to the FadeIn state.

Figure 4 shows all of these additional actions and transitions. The remaining empty cells are "should not occur" situations.
The table representation of a finite state machine can always be translated back into a graph, since the representations are equivalent. Figure 5 shows the graph representation of the completed table.

Gathering a list of state variables

After the state table and graph are complete, it is useful to review it one more time to gather a list of variables the machine will need to remember between events so that it can execute related actions in different cells. Your finite state machine will need state variables for the things in Listing 2.

Listing 2. Initial list of state variables

```javascript
currentState string value equal to one of the
state names currentTimer pointer to timer object, obtained when set, used to cancel
currentTicker pointer to ticker object, obtained when started, used to cancel currentOpacity
float that varies from 0.0 (invisible) to 1.0 (fully visible) lastCursorPosition floats obtained
from cursor events, used when an HTML Division element is created tooltipDivision pointer to
HTML Division element, set when created, used when faded, moved, or deleted
```

Though JavaScript variables are untyped, the values they contain are typed (that is, values of any type can be assigned to any variable). In this spirit, I chose names for the state variables and, in the commentary, indicated the types I expect to assign to them.
Ready to implement

With the completed state table and a list of the state variables, you are ready to implement your finite state machine. Stay tuned, as that is the focus of the next article. Remember, though, that development is an iterative process; you might need to return to the design phase ...
Related topics

- **Ajax: A New Approach to Web Applications**: Read Jesse James Garrett's article that coined the term Ajax.
- **Document Object Model (DOM) Level 2 Events Specification** (W3C, 2000): Refer to this spec for the authoritative definition of the DOM Level 2 event model.
- The *Gecko DOM Reference* (Mozilla): Get the authoritative definition of the object interfaces, including events, implemented by the Firefox browser.
- **HTML and DHTML Reference** (Microsoft): Refer to the authoritative definition of the object interfaces, including events, implemented by the Internet Explorer browser.
- Chapter 3.5 "Finite Automata" in *Compilers: Principles, Techniques, ad Tools* (Alfred V. Aho et. al., Addison-Welsley, 1986): Read a description of how finite state machines are applied to computer language compilers.
- Chapter 5 "Behavioral Patterns" in *Design Patterns: Elements of Reusable Object-Oriented Software* (Erich Gamma et. al., Addison-Welsley, 1995): Read about the State pattern for a discussion of implementing finite state machines.
- Chapter 13.25 "TCP State Machine" in *Internetworking with TCP/IP* (Douglas E. Comer, Simon and Schuster Company, 1995): Read to understand the finite state machine that underlies the Internet.
- **State Chart XML (SCXML): State Machine Notation for Control Abstraction** (RJ Auburn et. al., W3C): Read a proposed XML representation of finite state machines.
- Download the following browsers to test the FadingTooltip widget:
  - Mozilla Firefox browser
  - Microsoft Internet Explorer browser
  - Netscape Navigator browser
  - Opera browser
- Find more free downloads in the developerWorks Web Development Downloads and products area.
- Examples of browser tooltips and FadingTooltip widgets
- **HTML source code for examples of browser tooltips and FadingTooltip widgets**
- **JavaScript source code for FadingTooltip widget**