Reprint from SWITCHEDON Spring 2016. Download from goo.gl/2giOMp

FSM AT KS3

State diagrams are an important way of describing what computer systems can do. The last issue of SWITCHEDON highlighted a great way to introduce the concept; exploring the curious properties of a multi-sided hexahexaflexagon. A booklet describing the activity, by Teaching London Computing can be downloaded from bit.ly/1mdPFAi. A diagram of the transitions is a special kind of graph that can be thought of as a machine – a 'finite state machine'. The nodes of the graph are different states the flexagon can be in and the edges show what actions that can be taken to move between states. It describes the computations involved in flexing the flexagon.

Finite state machines (FSM) can serve as a vehicle to illustrate one of the core ideas of algorithms. Machines initiate actions based on their current state and on received inputs. As such, they offer another way to introduce programming. Rather than writing code, or assembling blocks of commands, simple programs can be expressed as diagrams.

Developed at ETH Zurich, Kara the ladybird lives in a world of trees, leaves and mushrooms. Children work in an easy-to-use environment without having to deal with the complexities of modern programming interfaces. One goal of the project was to ensure they could have a working program expressed through a FSM within an hour. A free download from bit.ly/1JDS0cG Kara comes with exercises that range from simple to very challenging. An excellent way for children to begin to appreciate that programing is but part of a much deeper science. *Roger Davies*

SOME REFLECTIONS ABOUT THE STATE THAT WE ARE IN

In the second of a new regular series, Greg Michaelson, Professor of Computer Science at Herriot-Watt University in Edinburgh, illustrates the value of state diagrams.

Suppose we want to design an interactive text editor. To begin with, let's concentrate on the functionality, so we're not concerned quite yet with how to use or implement it. Looking at lots of existing text editors to tease out their common features, we observe that they all provide the abilities to:

- open a *new* empty file;
- *open* an existing file;
- *close* a file;
- *save* a file;
- *edit* a file;
- *exit* the editor.

We also observe that:

- a file that isn't open can't be closed;
- a file that isn't open can't be saved;
- a closed file can't be changed;
- a changed file can't be closed before it's been saved;
- the editor can't be exited so long as a changed file hasn't been saved.

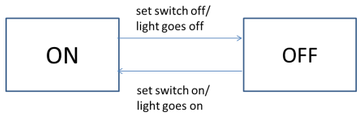
So a text editor can go through the following stages or *states*:

- ***ready*** – waiting for a file to be opened, or to exit the editor;
- ***open*** – waiting for an open file to be changed or closed;
- ***changed*** – waiting for a changed file to be altered further or to be saved.

And then the editor will change from one state to another depending on what the user wants to do next.

A very useful way to capture the transitions between the states of a system, without focussing on the low level details of what happens between each state, is with a ***state diagram***. State diagrams, which are closely related to ***finite state machines***, are also used in many engineering disciplines, for example to design control systems. In Computing, state diagrams are a key part of the Unified Modelling Language (UML), and are also used to design both GUIs and concurrent systems. Here, we'll use the UML notation where a state is represented as a labelled *box* and a transition between two states is a *directed arc* labelled with the cause of the transition (the *event*) and what happens during the transition (the *action)*.

For example, a state machine for a light switch might look like the diagram below:

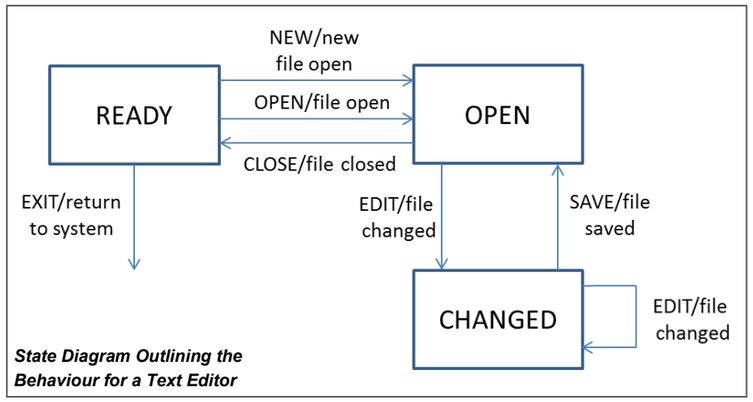


We can see that the light is in either a state where it's ON or a state where it's OFF. If the light is ON and the switch is set to off, the bulb goes out and the light enters the OFF state. If the light is OFF and the switch set to on, then the bulb goes on and the light is in the ON state.

You could imagine turning this into a GUI. The event of switching the light on or off might involve toggling a button and the associated action might be to change a label from, say, black (off) to white (on) and vice versa.

If we abstract away from this diagram we could use it as an outline design for any system with two states and transitions from each to the other.

So, let's now consider our text editor. The state diagram, showing the three states, ready, open and changed is shown in the box above right. Notice that we've named each event with a word. For example, we can consider the event EDIT to be a short hand for any change to the file.



State Diagram Outlining the Behaviour for a Text Editor

Think about the state diagram shown above. Does it manage to capture our intuitions about how our text editor should function?

Now, if we used this diagram as the basis for the development of a Graphical User Interface, we could consider putting the labels NEW / OPEN / CLOSE / SAVE / EXIT on buttons or pull down menus.

We could also restrict the display and only make an event option visible or enabled when its use is appropriate for the current state, that is only when there's an arc leading from a state labelled with the word for the event. Used in this fashion, the state diagram is also telling us how to control what the user can legally do.

A very nice thing about this approach is that we have separated out three things:

- the legal sequences of transitions – the state diagram;
- how each event is triggered by the user – the interface;
- how each action is realised – the implementation.

If we stick to our state diagram, we can make changes to the interface or the implementation independently of each other, making the realisation of the application much simple

This is reminiscent of the Model-View-Control (MVC) design pattern, articulated in the diagram (top right): the state diagram is the control, the interface is the view and the implementation is the model.

EXERCISES FOR EXPLORING THE IDEAS FURTHER

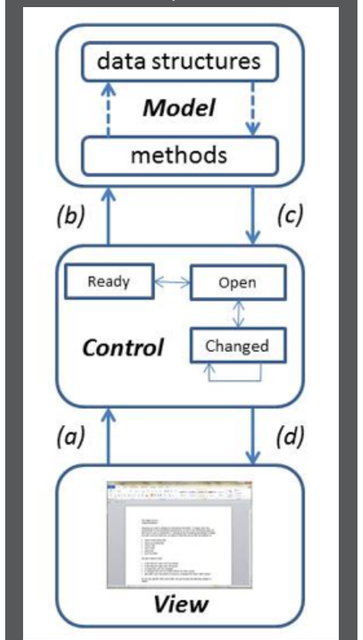
As an exercise, realise the text editor state diagram using your own choice of interface, say a command line or a GUI. Choose something simple to indicate the actions, for example displaying a message or changing a label when a state changes, but without actually implementing any of the actual actions. Once the interface behaves as you expect it to, implement the actions by developing appropriate procedures or methods that can be called as a result of selecting each event. Developing applications in this fashion allows the programmer to separate the implementation of the interface from the processing required for each action..

As another exercise, construct a state diagram that explains the operation of a media player, with controls to open, play, pause and stop media, and to exit. Then use the diagram to build an interface for the player.

Finally use your state diagram to construct an implementation of a slide show player for images. Here, because the image sequence continues to play until the user pauses or stops it, it's helpful to think about MVC and how to introduce interacting threads for each component.

MODEL – VIEW – CONTROL (MVC) DESIGN PATTERN

The MVC design pattern provides a way of cleanly separating out interface and behavioural concerns. The Model consists of the underlying data structures for the problem domain, typically manipulated via methods. The View is the user interface. The Control mediates and keeps track of the interactions between the View and the Model. For example, for our editor:



As the user interacts with the View, the View interacts with the Control (a), which in turn interacts with the Model (b). The Models uses its methods, manipulates information in the data structures and returns results via the Control (c) to the View (d) for the user to see.

MVC was originally developed for the early OO language Smalltalk. Note there are now lots of different and not necessarily consistent takes on MVC. More about design patterns in Java, including MVC in the tutorial at bit.ly/1P6mCtx.