The State Controller Pattern: An Alternative to Actions

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

ABSTRACT

A GUI application must enable and disable widgets as the state of the application changes. Bryan Oakley's notion of "actions" aids this by associating widgets with user actions; widgets associated with an action can be enabled and disabled as a group. This paper suggests a different pattern in which widgets are associated instead with the conditions for which they are enabled, thus greatly simplifying the problem of triggering widget state changes as the application state changes.

1. The Problem

As has often been noted at the Tcl Conference over the last few years, it is trivially easy to write a quick-and-dirty GUI using Tcl/Tk; the difficulty lies in giving the GUI all of the little professional touches. One of those professional touches is to make each GUI control be "grayed out" when the action it invokes is currently invalid.

For example, MS Word has a table editing toolbar. Some buttons on it create new tables; these will be enabled when the insertion point is in normal text, where a table can be inserted. Other buttons modify part or all of an existing table; they will be enabled only when the insertion point is within a table, or perhaps when some portion of the table is selected.

Anyone who has tried to create a polished GUI has had to address this issue; and has usually found that it is more work than seems reasonable.

2. The "Actions" Pattern

In his 2004 conference paper, "Actions: A Simple Implementation", Bryan Oakley describes the Actions pattern, his way of making the problem more tractable. [1] His insight is that there are often multiple ways to invoke any given action the user might wish to take. In a word processor, for example, the user might wish to copy the selection to the clipboard. He can:

- Enter a key command, **Control-C**
- Press the **Copy** button on the toolbar

- Select **Edit/Copy** from the Edit Menu

But although there are many input vectors, there is a single action underlying all of them: copy the selection. If there is no selection (or if the selected object cannot be copied), all of these vectors should be disabled. When something is subsequently selected, all of them should be enabled. Moreover, it should be possible to toggle them all at once. This is called setting the *validity* of the action.

The Actions pattern therefore has these characteristics:

- Each distinct action the user can do is represented as a `proc`-like pseudo-object in the application. For example,

  ```
  action define copy {} { event generate [focus] <<Copy>> }
  ```

- Each action's validity can be set:

  ```
  action enable copy
  action disable copy
  ```

- Each action can be invoked, which causes the action's body to be executed:

  ```
  action invoke copy
  ```

  If the action takes arguments, they are passed along with the action name.

- Each action can be associated with any number of buttons, menu items, key commands, and so forth:

  ```
  button .toolbar.copy -command {action invoke copy} ...
  action associate copy .toolbar.copy

  .menubar.editMenu add command     \
      -label   "Copy"               \
      -command {action invoke copy}
  action associate copy .menubar.editMenu "Copy"

  bind all <Control-c> [list action invoke copy]
  ```

- Setting the action's validity automatically enables or disables all related GUI controls.

- In Oakley's simple implementation, actions are managed by an "action manager" singleton; as he notes, they can also be implemented as explicit Tcl objects using an object framework like Snit.

As a result, the application logic responsible for setting action validity is nicely decoupled from the actual widgets that invoke those actions. This pattern has a number of advantages:

- The application logic is clearer, as it is written in terms of meaningful actions rather than arbitrary widgets.
- The application logic is shorter, as there are typically many fewer actions than GUI controls.
- The application logic is more stable with respect to GUI changes: the appearance of the GUI and the number of GUI controls per action can be completely changed without changing the application logic at all.
- The set of actions provides a vocabulary of commands that can be used to script the user interface, for test purposes or even for user scripting.

I refer to this as a pattern rather than as a library, because it usually makes sense to tailor the action manager implementation to the application.

3. Actions vs. Conditions

The reason why actions simplify the program logic is shown in the following diagram, which partitions the set of GUI controls in the program by the actions that they invoke:

a1	*a2*	*a3*	*a4*	*a5*	*a6*	*a7*	*a8*
g1	*g4*	*g6*	*g9*	*g12*	*g13*	*g15*	*g16*
g2	*g5*	*g7*	*g10*		*g14*		*g17*
g3		*g8*	*g11*				*g18*

In the diagram we have eight actions, *a1* through *a8*, and 18 GUI controls (menu items, buttons, etc.), *g1* through *g18*. By relating the controls with the actions, we now have less than half as many entities to enable and disable.

Now, it is likely that some of these actions are enabled and disabled under precisely the same conditions. Group the actions by these conditions, *c1* through *c4*:

c1			*c2*		*c3*	*c4*	
a1	*a2*	*a3*	*a4*	*a5*	*a6*	*a7*	*a8*
g1 *g2* *g3*	*g4* *g5*	*g6* *g7* *g8*	*g9* *g10* *g11*	*g12*	*g13* *g14*	*g15*	*g16* *g17* *g18*

Suppose that instead of associating the GUI controls with the actions, we associate them with the conditions for which they are valid. We now have fewer sets of GUI controls to deal with, and they are tied directly to the program logic (the condition) that sets their validity:

c1	*c2*	*c3*	*c4*
g1 g2 *g3 g4* *g5 g6* *g7 g8*	*g9* *g10* *g11* *g12*	*g13* *g14*	*g15* *g16* *g17* *g18*

This is essentially what the State Controller pattern does.

4. Notional Application

This section describes a notional application, to be used in examples in the remainder of the paper. The applications on which I am currently working have the following flavor:

- They are time-step simulations, in which the user prepares a scenario consisting of a number of entities of different kinds, and then allows simulation time to elapse, possibly intervening between time steps.

- They are document-centric. The scenario is a document that can be opened and saved.
- The application has distinct states; the user's actions are constrained by the current state.
  - **PREP:** Scenario Preparation. In this state the user can edit anything.
  - **PAUSED:** In this state, many aspects of the scenario are fixed. The simulation can be asked to starting running forward.
  - **RUN:** In this state, the user can watch the simulation execute, but cannot do much else other than pause it or return to PREP.
- Each entity has a set of attributes (i.e., a record structure). Consequently, the application has a browser for each kind of entity, allowing entities of that kind to be viewed, created, edited, and deleted.
- The application manages one simulation model at a time.
- The application has multiple windows, allowing multiple views on the model. All windows are identical in their capabilities (as with multiple editor windows on the same document). In particular, all of the various browsers are available as tabs in each of the windows.
- The application has a modular architecture based on Snit widgets. [3] Each window contains a variety of components, each implemented as a Snit widget.

Our notional application is a simulation of cars, roads, and gas stations:

- Roads and gas stations must be created and edited during PREP.
- Individual cars can be created and edited during PREP and while PAUSED.

5. State Controllers

The Actions pattern partitions the set of GUI controls by the actions they invoke. The basic notion of the State Controller pattern is to partition the set of GUI controls not by the actions they invoke, but by the conditions for which they are valid.

A state controller is a Snit object that controls the `-state` of some set of objects according to a `-condition` expression. If the condition is true, the `-state` of the objects is set to `normal`; otherwise, it is set to `disabled`. The API is straightforward:

`statecontroller` *`name`* `-condition {`*`expression`*`}`
: Create a new state controller with a given condition.

`name` `control` *`object ?key value ?key value...??`*
: Ask state controller *name* to control the `-state` of the named *object*. The optional arguments represent a dictionary of values to be used in the `-condition` expression.

  If *object* is a list of length 2, then the first token is assumed to be the name of a Tk menu widget and the second is the label of a menu item.

`name` `update ?`*`object...`*`?`
: Ask state controller *name* to update the `-state` of all of the objects it controls. Optionally, it can update only the named objects.

Once the state controller has been created and the GUI controls have been associated with it, all that remains is to invoke the controller's `update` method on events that might cause the controller's `-condition` value to change.

Note that there is no command to forget a controlled object. If a controlled object no longer exists, the state controller silently forgets it.

5.1 Example: Adding an Entity

In our notional application, road and gas station entities can be created only while the simulation is in the PREP state. Each kind of entity has a browser, and each browser has an "Add" button.

First we define the state controller; its condition, of course, is simply that the simulation state is PREP:

```
    statecontroller inprep -condition {[sim state] eq "PREP"}
```

Because this condition applies to GUI controls in different components, we define `inprep` at application scope.

Next, in each browser we ask `inprep` to control the "Add" button:

```
    # Associate a widget with the statecontroller
    button $win.add -text "Add" -command {...}
    inprep control $win.add
```

When the simulation state changes the simulation notifies the `inprep` object:

```
    inprep update
```

The controller then sets the `-state` of all controlled widgets in one fell swoop. That is all there is to it. The advantages are plain:

- Instead of defining "addroad" and "addstation" actions we define a single state controller.
- Associating the GUI controls with the state controller is no more work than associating them with an action.
- Instead of having code in every browser to subscribe and respond to simulation state changes, we call `inprep update` once when simulation state changes..

5.2 Example: Editing or Deleting an Entity

Each entity browser also has Edit and Delete buttons, which are valid only if the simulation is in the "PREP" state *and* an entity is selected in the browser. Pressing the button pops up a dialog used to edit the entity's attributes.

This is a difficult case to implement with the Actions pattern, because we can have multiple instances of a particular browser. For example, we might have two windows open, each displaying the gas station browser. Different gas stations might be selected in the two browsers, so that the Edit button is valid in one but not in the other. As a result, we would need to have two actions (or, more generally, *N* actions), one for each gas station browser. This is doable, certainly, but it smells funny: we have two "actions", but only one real action being performed.

The State Controller pattern takes this in stride. All we need do is to parameterize the state controller's `-condition` with the name of the browser:

```
    statecontroller inprepWithEntity -condition {
        [sim state] eq "PREP" &&
        [llength [$browser selection]] == 1
    }
```

We might also need to add a method to the browser widget itself; but as we are using Snit this is straightforward.

The value of `browser` is specified when a widget is controlled:

```
button $win.edit   -text "Edit"   -command {...}
button $win.delete -text "Delete" -command {...}

inprepWithEntity control $win.edit   browser $win
inprepWithEntity control $win.delete browser $win
```

Now, `inprepWithEntity` must be updated on two events: when the simulation state changes, and when the browser's selection changes:

```
# When sim state changes (In the "sim" module):
inprepWithEntity update

# When selection changes (In the browser component):
inprepWithEntity update $win.edit $win.delete
```

Changes to the browser's selection only affect the widgets that are part of that browser; consequently we ask the controller to update them specifically. The controller then evaluates its condition for those widgets alone. This is a minor performance hack, and can reasonably be omitted.

The menu items for editing and deleting entities only depend on the simulation state, and so can be controlled by the `inprep` controller.

5.3 Widgets with Unusual States

Most Tk widgets have two values for `-state`: `normal` and `disabled`. A few have additional states, such as `readonly`. If `normal` and `disabled` are not the correct pair of states for a given widget, the easiest solution is to wrap the widget in a `snit::widgetadaptor` and let the wrapper translate `normal` and `disabled` into the desired pair of values.

6. Triggering State Controller Updates

It is unpleasant to sprinkle calls to a state controller's `update` method around the code to catch all of the events that make a state controller's `-condition` change. The code is clearer if the module that defines the state controller can also subscribe to all of the relevant global events, which implies the need for a non-GUI application-level event mechanism. Although Tk virtual events can be used to do this job, they are unsatisfactory. Instead, we use a mechanism we call the `notifier`. Any module can send notifier events at any time, without declaration:

```
notifier send ::sim <State>
```

Here, `::sim` is the module sending the event and `<State>` is the event being sent. (Events can also include arguments.) Any number of modules can subscribe to this event from this sender:

```
    notifier bind ::sim <State> inprep {
        inprep update
    }

    notifier bind ::sim <State> inprepWithEntity {
        inprepWithEntity update
    }
```

This allows the definition of the state controller and the global calls to `update` to be co-located:

```
    statecontroller inprep -condition {[sim state] eq "PREP"}
    notifier bind ::sim <State> inprep {inprep update}
```

This combination appears often enough that our implementation of `statecontroller` supports it directly:

```
    statecontroller inprep \
        -condition {[sim state] eq "PREP"} \
        -events    {::sim <State>}
```

7. Combining State Controllers with Actions

Suppose that instead of using a singleton action manager, as described in Section 2, each action was implemented as a Snit object with a `-state` option. One could then implement a three-tier architecture: GUI controls are associated with action objects, and action objects are controlled by state controllers. This preserves the use of actions as a kind of scripting language for the application. In our application domain, however, we have approached it differently.

We have the notion of an *order*, a user input that changes the simulation model. Looked at another way, any change that needs to be saved as part of the scenario document is made by an order. In our example, we would have orders to add, edit and delete roads, gas stations, and cars, along with many other operations. Like actions, orders are a kind of managed pseudo-proc. Unlike actions, orders are non-GUI.

Orders provide the application with the same scripting ability as actions, and at a level better suited to the application domain. (They also provide a basis for multi-level undo/redo.)

The primary variable that determines whether an order can be validly sent or not (assuming its parameters are valid) is the simulation state, and consequently the order manager handles this explicitly. Each order "knows" the simulation states in which it is valid. The order manager subscribes to the "`::sim <State>`" notifier event, and sends the "`::order <State>`"

event when order validity changes. Thus, we can write state controllers like this:

```
    statecontroller orderValid \
        -condition {[order valid $order]} \
        -events    {::order <State>}

    statecontroller orderValidWithEntity \
        -condition {
            [order valid $order] &&
            [llength [$browser selection]] > 0
        } \
        -events {::order <State>}
```

When we associate a GUI control with the latter of these state controllers, we need to specify the name of the order along with the name of the browser. Thus, we can edit the list of states for which the order is nominally valid as part of the order definition, and all of the GUI controls will be enabled and disabled accordingly, with no other code changes.

In short, instead of using actions to mediate between state controllers and GUI controls, we are using state controllers to relate orders to GUI controls.

8. Key Commands

Oakley's action manager implementation disables key bindings by making `action invoke` a no-op when the action is disabled. Because a state controller can control many widgets invoking many different operations, it has no notion of an `invoke` method. In order for a state controller to control a key binding, then, it would be necessary to define the key binding as an object with a `-state` option. It would be easy to implement a Snit type that wraps key bindings in this way, but to date we have not felt the need to do so.

9. State Controllers in Practice

The example in Section 3 suggests that a complex application could require many fewer state controllers than actions to do the same work, while greatly reducing the application logic in the GUI components that have controlled widgets.

In the most recent release of the project on which I am currently engaged, the main application has approximately 75 orders. These orders are invoked by approximately 150 GUI controls **per application window**. The state of these 150**N* GUI controls is managed by just 8 state controllers defined at the application level. The code in the components that contain these GUI controls is limited to the following:

- Associating the controls with the relevant state controller

- Invoking the state controller's `update` method when the component's state changes (only if needed).

We add new state controllers very rarely, and the rest of the time we do not think much about them; they just work.

In short, enabling and disabling GUI controls is no longer more work than seems reasonable.

10. References

[1] Oakley, Brian, "Actions: A Simple Implementation", 11th Tcl/Tk Conference, http://www.tcl.tk/community/tcl2004/Papers/BryanOakley/oakley.pdf.

[2] Oakley, Brian, "Actions", http://wiki.tcl.tk/11505

[3] Duquette, William H., "Snit's Not Incr Tcl", http://www.wjduquette.com/snit.

11. Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.