SwingStates: adding state machines to the swing toolkit

Caroline Appert, Michel Beaudouin-Lafon

SwingStates: Adding State Machines to the Swing Toolkit

Caroline Appert & Michel Beaudouin-Lafon
LRI (Univ. Paris-Sud & CNRS) / INRIA Futurs
Bât. 490, Univ. Paris-Sud, 91405 Orsay, France
appert@lri.fr, mbl@lri.fr

ABSTRACT

This article describes *SwingStates*, a library that adds state machines to the Java Swing user interface toolkit. Unlike traditional approaches, which use callbacks or listeners to define interaction, state machines provide a powerful control structure and localize all of the interaction code in one place. *SwingStates* takes advantage of Java's inner classes, providing programmers with a natural syntax and making it easier to follow and debug the resulting code. *SwingStates* tightly integrates state machines, the Java language and the Swing toolkit. It reduces the potential for an explosion of states by allowing multiple state machines to work together. We show how to use *SwingStates* to add new interaction techniques to existing Swing widgets, to program a powerful new Canvas widget and to control high-level dialogues.

ACM Classification: D.2.2 [Design tools and Techniques]: User Interfaces; H.5.2 [Information Interfaces and Presentation]: User Interfaces – Graphical User Interfaces.

General terms: Design, Human factors

Keywords: toolkit, state machine, Java Swing, widget

INTRODUCTION

Programmers today rely on user interface toolkits to program graphical user interfaces, assembling standard widgets to create the user interface and linking them to the rest of the application through callbacks or listeners. Such interactive applications are difficult to debug and maintain due to the difficulty of following the flow of control [8]. Moreover, adding new widget classes is complicated. As a result, mainstream applications rarely feature novel interaction techniques. Some toolkits, e.g., subArctic [6], address this issue, but are not used by most developers.

SwingStates takes a different approach by extending Java

Swing, an existing, widely used toolkit. Breaking away from the traditional callback model, *SwingStates* uses state machines as a powerful control structure for programming interaction. Since Newman's seminal work [10], state machines have often been used to describe interaction but rarely to actually program it [7, 13].

The strength of *SwingStates* is the tight integration between state machines, the Java language and the Swing toolkit. We allow multiple machines to work together, thus reducing the potential for an explosion in the number of the states. We describe the syntax and semantics of *SwingStates* and illustrate how it combines power and simplicity when programming advanced interaction.

CODING STATE MACHINES IN JAVA

A state machine (Fig. 1) consists of a set of *states* and a set of *transitions* labeled with *events*. Each transition connects an *input state* to an *output state* and may have an associated *guard* and *action*. Conversely, each state has a set of *output transitions* and a set of *input transitions* and may also have an *enter* and a *leave action*. Only one state is active at any one time. When an event occurs, the first output transition of the active state that is labeled with that event and whose guard evaluates to true, is *fired*. Firing a transition executes three actions in sequence: the *leave action* of the active state; the action associated with the transition being fired; and the *enter action* of the output state. The output state is now the new active state.
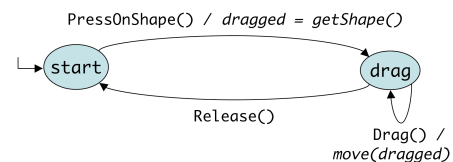
Figure 1: A simple state machine for dragging objects. States are circles; transitions are arrows, labeled with events (roman font) and actions (italics). This example includes neither enter and leave actions nor guards.

SwingStates implements state machines using Java's *inner classes*, which allow new subclasses to be declared in-line wherever a new (unique) object of that class is needed. The inner class is anonymous and has a single instance. Beyond the in-line syntax, inner classes have the advantage that their instances can access the fields and

methods of the object within which they are declared. Java Swing uses this syntax to attach listeners to widgets. Fig. 2 illustrates a classic Swing idiom where an instance of an anonymous subclass of ActionListener is created and passed to the quitButton's addActionListener method. The new subclass redefines the actionPerformed method so as to quit the application when the button is clicked.

```
1 quitButton.addActionListener(
2    new ActionListener(){
3       public void actionPerformed(ActionEvent e)
4              { System.exit(1) ; }
5    }) ;
```

Figure 2: A button to quit an application.

SwingStates uses inner classes to declare each state within the state machine and to declare each transition within its input state. Fig. 3 shows *SwingStates'* version of the state machine in Fig. 1.

```
1  StateMachine   sm = new StateMachine() {
2    SMShape dragged = null;
3    public State  start  = new State() {
4      Transition  dragOn  =
5        new PressOnShape(BUTTON1,  "drag") {
6          public void action() {
7            dragged = getShape();
8          }
9      };
10   };
11   public State  drag  = new State() {
12    Transition   drag  = new Drag(BUTTON1,  "drag") {
13      public void action() {
14        move(dragged);
15      }
16    };
17    Transition   dragOff  =
18      new Release(BUTTON1,  "start") { } ;
19   };
20 };
```

Figure 3: The SwingState state machine for Fig. 1.

Each transition is declared by instantiating a class that corresponds to the event labeling it. *SwingStates* provides a large and extensible set of event classes, including mouse events (Fig. 3: lines 5, 12, 18), keyboard events, timer events and virtual events (Fig. 7 & 8 feature virtual "color" events). Some transitions include a guard, e.g., PressOnShape (line 5, Fig. 3) corresponds to a mouse press on any shape of the Canvas widget described below.

A transition action is specified by overriding the action method (lines 6-8 and 13-15, Fig. 3). Transition guards and state enter and leave actions are specified by overriding similar methods. Note that the scoping rules of inner classes allow these methods to access the enclosing objects, such as the dragged variable of the state machine (declared at line 2 and used at lines 7 and 14, Fig. 3).

The transition's output state is always the last argument of its constructor and is specified using a string, e.g., "drag" (line 5, Fig. 3) refers to the state drag (line 11). We must

use a string rather than a direct reference to the field because it is not initialized at the time the constructor is called. We use the Java reflection interface to transform the string into a reference to the field the first time the state machine is used.

Like Interactors [9] and SubArctic's dispatch agents [6], *SwingStates'* machines externalize the management of interaction into separate objects. Interactors however use a single generic state machine, which may not handle all novel techniques, while SubArctic does not provide explicit support to program new dispatch agents.

ATTACHING STATE MACHINES TO UI OBJECTS

A powerful way to establish the link between a state machine and the target objects it controls is to use *tags* similar to those in the Tk toolkit [12]. Tags can be associated with any Swing widget and any shape of our Canvas widget described below. Many *SwingStates* transition classes, such as Press, have a variant, such as PressOnTag, that only fires when the event occurs on an object with the specified tag. For example, selectionTag could be added to currently selected objects. The transition PressOnTag(selectionTag) would then fire only when a mouse button is pressed on an object with selectionTag. Tags may also have names, so this could also be specified as PressOnTag("selected").

SwingStates defines two kinds of tags: *extensional tags*, which are added to or removed from objects explicitly, and *intentional tags*, which are specified using a predicate, e.g., all objects with a blue background. For extensional tags, subclasses can override methods that are called when the tag is added or removed. For example a SelectionTag class could specify that the target object changes color when the tag is added and restores its original color when the tag is removed. Tags also implement some methods of their target objects for easily manipulating the group of tagged objects. For example, the method call selectionTag.setBackground(blue) changes the color of all the objects with this tag.

USING STATE MACHINES

This section shows how multiple state machines can work both independently and together to create rich interactions from simple building blocks.

A Generic Crossing Interface

SwingStates can attach state machines to regular Swing widgets to extend or redefine their behavior. This example (Fig. 4) modifies the behavior of Swing buttons so they are selected by crossing rather than clicking. When attached to a button widget, the state machine activates the button as soon as the cursor finishes crossing the widget with the mouse button depressed. Note that several buttons can be activated in a single stroke, as in CrossY [1]. Other crossing interactions, such as CrossY's

scrollbars, could be implemented as well in a single state machine by using the class names of the widgets as tags. In order to control the state explosion problem, they could also be implemented as separate state machines.

```
1 JStateMachine    cross = new JStateMachine() {
2   public State out = new State() {
3       Transition   enter  = new EnterOnTag
                        ("javax.swing.JButton", "in") { };
4   };
5   public State  in = new State() {
6       Transition  leave  = new Leave("out")  {
7           public void action() {
8               ((JButton)getComponent()).doClick();
9           }
10      };
11 };
// attach this state machine to the quit button of Fig. 2
12 cross.attachJComponent(quitButton);
```

Figure 4: A crossing interface for Swing buttons.

One problem with the above implementation is the lack of an ink trail. *SwingStates* can attach a state machine to the *glasspane*, a feature of Swing that creates an overlay plane above the widgets in a window, in order to draw the ink and erase it when the pen is up. Fig. 5 shows another example that uses the glasspane to enter a numeric value in an entry field using a joystick-like interaction.

Figure 5: A joystick style of interaction for text entry.

A Canvas Widget

While some interaction techniques can be added to existing widgets, as shown above, many require novel widgets. Swing does not have a Canvas widget similar to that of Tk [12] so the programmer has to use the lower-level Java2D library to create new widgets. *SwingStates* includes a Canvas widget [2] inspired by an extended version of the Tk canvas widget called GmlCanvas [4]. Each canvas holds a display list of shapes, including simple or arbitrary paths, text strings, images and even Swing widgets. Each shape can have a geometric transform, a parent shape and a clipping shape. Shapes can be tagged and state machines can be attached to a canvas, to individual shapes, and even to tags. Several state machines can be active at once, running in parallel.

We used an earlier version of *SwingStates'* Canvas [2] in a Master's level computer science course where students implemented a wide variety of interaction techniques, including toolglasses, magnetic guidelines and side views. Unlike our attempts in previous years with other toolkits, all students completed their projects with little or no help, demonstrating the power and simplicity of the canvas and state machines to implement advanced interactions.

Swing Pie Menu

Figures 7 and 8 show how to implement a pie menu (Fig. 6) using a canvas, the glasspane and two state machines.
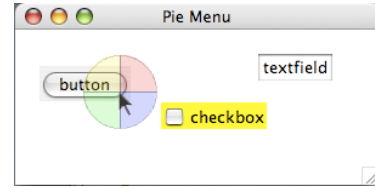

Figure 6: A pie menu that changes the color of other arbitrary Swing widgets: button, checkbox, text field.

A canvas containing the pie menu runs in the glasspane over the content pane that contains arbitrary Swing widgets (button, checkbox, etc.). The smMenu state machine (Fig. 7) is attached to the canvas. It controls menu invocation (show/hide menu, lines 11 and 25) and selection of an item (line 18). When an item is selected, it sends a color event (line 21) to the smWidget state machine, which is attached to the content pane. smWidget (Fig. 8) selects a widget when a button is pressed (line 4, Fig. 8) and changes the background color of the selected widget when it receives a color event (line 10, Fig. 8).

```
// ColorTag are designed to be added to each menu item.
1 class ColorTag extends CExtensionalTag {
2     Color color;
3     ... // constructor
4 }

// Color events are sent by smMenu to smWidget.
5 class ColorEvent extends VirtualEvent {
6     Color color;
7     ... // constructor
8 }

//  The state machine that manages the pie menu
9  smMenu = new CStateMachine(canvas)   {
10    public State menuOff  = new State() {
11      Transition show = new Press(BUTTON1,"menuOn"){
12        public void action() {
13          showMenu(getPoint());
14        }
15      };
16    };
17    public State  menuOn  = new State() {
18      Transition command  = new ReleaseOnTag
                (ColorTag.class, BUTTON1, "menuOff") {
19        public void action() {
20          Color c = ((ColorTag)getTag()).color;
21          smWidgets.processEvent(new ColorEvent(c));
22          hideMenu();
23        }
24      };
25      Transition hide = new Release
                            (BUTTON1, "menuOff") {
26        public void action() {
27          hideMenu();
28        }
29      };
30    };
31 };
```

Figure 7: Invoking the pie menu and selecting items.

This example illustrates the use of two communicating state machines: `smMenu` handles the pie menu while `smWidgets` handles the target objects. It makes it easy to replace the pie menu by, e.g., a toolglass, or, conversely, to use the pie menu with other graphical objects. This approach cleanly separates the interaction technique from the objects of interest, in the spirit of instrumental interaction [3]. Running state machines in parallel also reduces the state explosion problem without resorting to more complex or more general models such as hierarchical state machines [5] or PPS [11].

```
// "colorable" widgets must be attached to this state machine
1  smWidgets = new JStateMachine()  {
2    JComponent picked;
3    public State  noSelection  = new State() {
4     Transition select = new PressOnComponent()   {
5       public void action() {picked = getComponent();}
6     };
7    } ;
8    public State  selection  = new State() {
9     Transition deselect
              = new Event("cancel",  "noSelection") {} ;
10    Transition color
              = new Event("color",  "noSelection") {
11      public void action() {
12        ColorEvent e = (ColorEvent)getVirtualEvent();
13        if (selected) picked.setBackground(e.color);
14      }
15     };
16    };
17 };
```

Figure 8: Selecting and coloring the widgets.

IMPLEMENTATION
SwingStates has 7250 lines of source code (excluding comments and documentation); the library is 144 Kb. We have not observed any significant performance overhead when compared with plain Swing applications.

While we could have implemented a graphical editor for state machines, we have observed that such tools are rarely used by programmers who prefer compact, textual constructs. *SwingStates* builds on programmers' existing skills and Java's features to condenses the complete state machine into one block of code that can readily be run and debugged. By contrast, a graphical form would require the actions' code to be separate from the graphics, as well as specific tools for execution, debugging, and storage of state machines.

CONCLUSION AND FUTURE WORK
SwingStates (http://insitu.lri.fr/SwingStates) is a novel way to use state machines to program advanced interaction techniques with the popular Java Swing toolkit. The tight integration between the state machines, the Java programming language and the Swing toolkit combines power and simplicity, eliciting a programming style where multiple state machines can work independently or in combination to create rich interactions. We used

SwingStates successfully to teach graduate and undergraduate students and within our group to explore novel interaction techniques. We now plan to support bimanual interaction, explore distributed interfaces and apply a similar approach to other contexts.

REFERENCES
1. Apitz, G. & Guimbretière, F. (2004) CrossY: a crossing-based drawing application. In *Proc. ACM Symposium on User Interface Software and Technology*, UIST '04. ACM Press, pp 3-12.

2. Appert, C. & Beaudouin-Lafon, M. (2006) SMCanvas: augmenter la boîte à outils Java Swing pour prototyper des techniques d'interaction avancées. in *Proc. Conférence Francophone sur l'Interaction Homme-Machine*, IHM '06. ACM ICPS, pp 99-106.

3. Beaudouin-Lafon, M. (2000) Instrumental interaction: an interaction model for designing post-WIMP user interfaces. In *Proc. ACM Conference on Human Factors in Computing Systems*. CHI '00. ACM Press, pp 446-453.

4. Bérard, F. GmlCanvas. http://iihm.imag.fr/projects/gml

5. Blanch, R. & Beaudouin-Lafon M. (2006). Programming interaction with hierarchical state machines. *In Proc. ACM Conf. on Advanced Visual Interfaces*. AVI'06. pp 51-58.

6. Hudson, S. E., Mankoff, J., and Smith, I. (2005) Extensible input handling in the subArctic toolkit. In *Proc. ACM Conference on Human Factors in Computing Systems*. CHI '05. ACM Press, pp 381-390.

7. Jacob, R.J., Deligiannidis, L., and Morrison, S. (1999) A software model and specification language for non-WIMP user interfaces. *ACM Trans. Computer-Human Interaction*, 6(1):1-46.

8. Myers, B.A. (1991) Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. ACM Symposium on User interface Software and Technology*. UIST '91. ACM Press, pp 211-220.

9. Myers B.A. (1990) A New Model for Handling Input. *In ACM Trans. on Information Systems*, 8(3):289-320.

10. Newman, W. M. (1998) A system for interactive graphical programming. *In Seminal Graphics: Pioneering Efforts that Shaped the Field*. ACM Press, pp 409-416.

11. Olsen, D. R. (1990) Propositional production systems for dialog description. In *Proc. ACM Conf. on Human Factors in Computing Systems*. CHI '90. ACM Press, pp 57-64.

12. Ousterhout, J. K. (1994) *Tcl and the Tk Toolkit*. Addison-Wesley.

13. Wellner, P. (1989) Statemaster: A UIMS based on statechart for prototyping and target implementation. In *Proc. ACM Conference on Human Factors in Computing Systems*. CHI '89. ACM Press, pp 177-182.