Novel Widgets and Toolkits For Creating Them

Rachel Nehmer

1. Introduction

I explored three different toolkits (SwingStates, subArctic, and MAUI) that provided custom widgets. Two of these toolkits, SwingStates and subArctic, also provided interaction techniques that are aimed at allowing the programmer to develop novel widgets. The third toolkit, MAUI, is a toolkit that provides the developer of groupware applications with some built in groupware widgets and network support. All three toolkits are Java libraries and extend some of the Swing widgets while adding some new widgets as well.

2. The SwingStates Toolkit

SwingStates adds state machines to the tools available for Java GUI developers.[2] The state machine is used as a control structure for keeping all of the interaction code in one place. What is most appealing about SwingStates is that you can design and implement an interaction technique separate from the widget that will be using it. Also, once you implement your state machine, attaching it to UI elements is very flexible. The following examples and illustrations are from the paper "*SwingStates*: Adding State Machines to the Swing Toolkit" by Caroline Appert & Michel Beaudouin-Lafon.

2.1 Ideas Borrowed From Tk

SwingStates borrows some things from Tk that strengthen the programmers ability to use the interaction techniques they may develop. First of all, it implements a canvas that behaves similar to Tk's canvas. You have a display list of shapes, each shape can have a geometric transform, a parent shape, and a clipping shape. The canvas makes it easier to draw and keep track of arbitrary shapes. Secondly, SwingStates has the ability to add tags to canvas shapes as well as any Swing widget. Because you can add state machines to tags, you really have a lot of flexibility in creating interface components separate from the state machines. I plan on exploring this toolkit more to see how easy it really is to create my own novel widget.

2.2 The State Machine

I think the hardest part of creating a novel widget is coming up with a useful and interesting idea. Once you have formulated the idea and can translate the user interaction into a state machine then you have done the hardest part. It is much easier to instantiate the idea and I will describe the process briefly.

The easiest way to implement a state machine is to first draw out the state machine that represents your interaction technique. I will use the example of creating a state machine for dragging objects. I don't need to know what objects these will be beforehand.

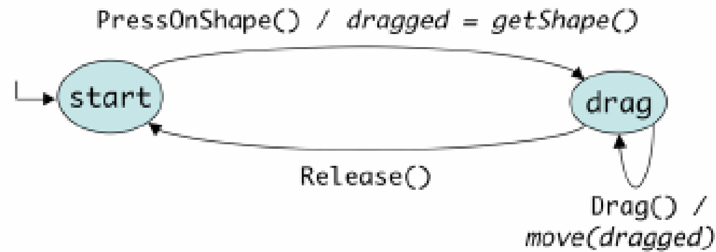


Figure 1: A simple state machine for dragging objects. States are circles; transitions are arrows, labeled with events (roman font) and actions (italics). This example includes neither enter and leave actions nor guards.

Once you have drawn out the state machine it becomes a matter of translating the diagram into code. SwingStates uses inner classes heavily so you will have a StateMachine class that contains one or more State classes with each of the State classes containing one or more Transition objects. The Transition objects refer to other inner State classes and this is done by using a string in the constructor and then using Java reflection to transform the string into a reference. At first glance the code for the above example seemed a little funny but once I understood what was going on, it was very appealing. I liked the tidy nature of the code.

Below is the SwingState implementation of the state machine from figure 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 };
```

These particular Transition objects are from the set of canvas Transitions. The Transitions can cause an action to be performed or just transition from one state to another (both types are seen above). As you can see from the code above, the state machine is a self contained piece of code for implementing interaction.

2.3 Attaching the State Machine
The state machine can be attached to a tag, a Swing widget, or a canvas widget (includes shapes). You can either attach the state machine directly to the component by using the appropriate attach method (i.e. AttachJComponent) or you can add the state machine as a listener of a component using the method addAsListenerOf(JComponent). There are similar methods for tags and canvas widgets.

2.5 Some Novel Widgets
A list of check boxes that are selected by a crossing interface. Moving the cursor across a check box will select or deselect it and the default of clicking on the box can still be supported.

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

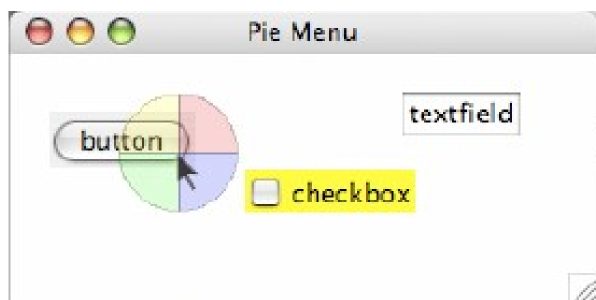


Figure 6: A pie menu that changes the color of other arbitrary Swing widgets: button, checkbox, text field.

2.6 Conclusion of SwingStates
I think SwingStates is a good toolkit because it is focused on improving the process of programming interaction. The state machine gives us a way of representing the full interaction process and separating it from the components so that the component does not need to worry about keeping track of state. This separation makes for good programming and nice looking code.

3. The subArctic Toolkit

There are some similarities between subArctic[3] and SwingStates in the way they look at interaction. Both toolkits use the state machine as a model for describing user input. They differ in the way that they implement the model. subArctic uses dispatch agents to handle different types of input. The examples and illustrations are from the paper "Extensible Input Handling in the subArctic Toolkit" by Scott E. Hudson, Jennifer Mankoff, and Ian Smith.

3.1 Dispatch Agents

Dispatch agents work by defining an input protocol. If a component wants to be called by the dispatch agent it needs to implement the Java interface defining the protocol. The protocol should represent an input pattern. The idea is to define the full interaction sequence in the dispatch agent so that, once again, the component does not need to keep track of what happened beforehand. All the component needs to do is implement the protocol interface and not worry about the user events. This is useful because the user may change input devices and the only thing that would need to change is that we would need a dispatch agent that interprets the devices input.

3.2 Picking

One thing subArctic does that is unique is that it allows the programmer to have control over what object is picked when a selection occurs. This is useful because there may be more than one object under the cursor and you may want to pick both of them. Normally, the highest priority object (the leaf node in the interactor tree) is picked but we could have the parent of this node be picked instead.

The following is a container that changes the picking order by adding itself to the picking collection and then adding all of its children. Any widget can be added to this shadow container and when the user clicks on any widget in the collection, all of the widgets are picked and a "shadow" is drawn underneath to enhance the feel of dragging the widgets.

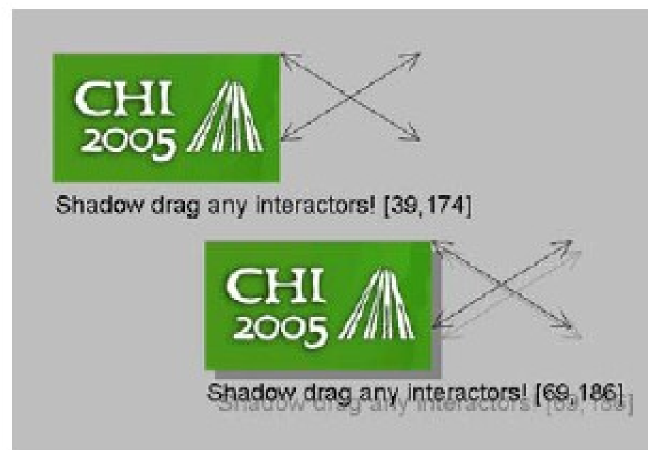


Figure 1. Shadow-drag container interaction

3.3 Conclusion of subArctic

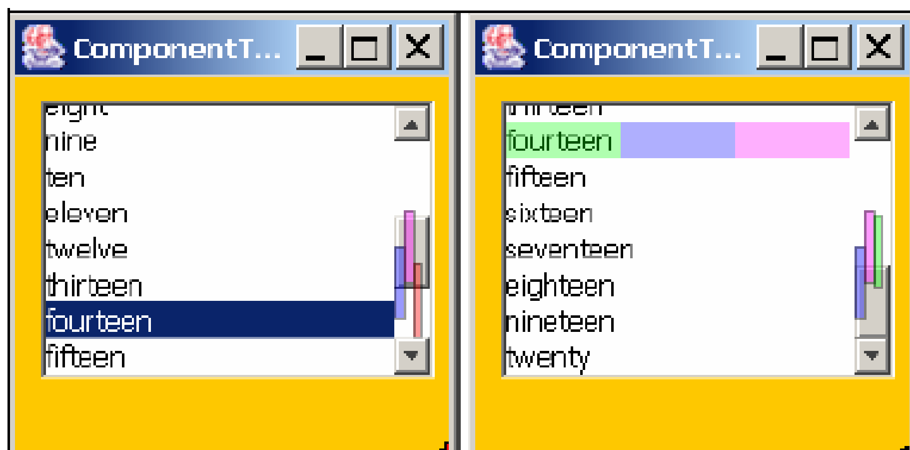
The subArctic toolkit was hard to evaluate because the website was a little out of date and many of the demo applets failed to load for me. I think the ideas behind the toolkit are good but I'm not sure how practical it is to use the toolkit for real development.

4. The MAUI Toolkit

I looked at both the MAUI (Multi-user Aware UI) toolkit[1] and Groupkit and there was a good deal of overlap between the two. Both are toolkits for creating Groupware applications but Groupkit uses Tcl/Tk and was developed much earlier than MAUI. MAUI uses many of the ideas from Groupkit but also extends a large number of Java Swing widgets which allows Java programmers to use components they are already familiar with.

4.1 Extending Java Swing Widgets

What MAUI does with Swing Widgets is attach additional event listeners that respond to both awareness events (the cursor moving into a button) and action events (button pressed). These additional listeners are responsible for sending relevant information out to the other users. The widgets can be configured by the user to enable, disable, or modify awareness. The default values for widget awareness can be set by the programmer. Many of the widgets have more than one form of awareness. For example, there is the option of a shared or multi-user scrollbar. In the shared version, when a user moves the scrollbar it is updated in all user views. In the multi-user scrollbar, when a user moves the scrollbar a highlighted version of their scrollbar and selection is drawn in all user views.



The nice thing about extending the Java Swing widgets is that you can use a combination of shared components and single user components. All you have to do is add the appropriate event listeners to the shared widgets. I think the use of familiar Java Swing widgets is the MAUI toolkit's strongest feature.

4.2 Widgets Borrowed

MAUI borrows the telepointer widget and the participant list widget from earlier groupware toolkits. The telepointer is especially useful because much is conveyed in gestures and being able to gesture in groupware is very important. One of the papers I read explained that when observing a group of people at a whiteboard, more gesturing occurred than actual writing or drawing on the board.

4.3 Java Beans

Packaging the components as Java Beans allows the toolkit to be used in IDEs like JBuilder. The video demonstration of the MAUI toolkit is done in the JBuilder IDE and it seems very easy to use. I do not use an IDE so I did not attempt to integrate the MAUI toolkit but it would be interesting to see how long it would take to develop something like a shared whiteboard.

4.4 Conclusion of MAUI

I think MAUI has good potential and I hope the people working on it continue to do so. MAUI does all of the work relaying awareness information over the network which gives programmers some free time to come up with interesting applications. This is another toolkit that I would really like to try using.

5. References

[1] Appert, C. & Beaudoin-Lafon, M. (2006) SwingStates: Adding State Machines to the Swing Toolkit. In *Proc. ACM UIST '06*, pp 319-322.

[2] Hill, J. & Gutwin, C. Awareness Support in a Groupware Widget Toolkit. Computer Supported Cooperative Work. **13**(5-6):539-571, 2004.

[3] 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.