Lafros GUI-App: a monitoring and control-oriented Scala-Swing application framework

Rob Dickens
Latterfrosken Software Development Limited, 32 Bradford St, Walsall, West Midlands, UK, WS1 3QA
rob.dickens@lafros.com

Abstract

Lafros GUI-App offers a lightweight means of simplifying the task of writing monitoring and control-oriented desktop user interfaces in Scala. It is a complete rewrite of an existing Java framework, JUICe.app, in Scala. The principal facilities provided are, a means to run the same code as either an application or applet, an environment for executing abstract commands, and a specialised label component for displaying values to be monitored. After first relating the software's development history, the paper describes the facilities provided and how to use them. The benefits of using Scala and GUI-App versus Java and JUICe.app are then considered, including being able to write code which is more concise and declarative in style. Finally, an issue arising from the introduction of dependencies on the Scala libraries is addressed to conclude.

Keywords Scala, application framework, GUI, monitoring and control

1. Introduction

Lafros GUI-App[1] is a lightweight application-framework based on Scala-Swing, intended to simplify the task of writing monitoring and control-oriented user interfaces for the desktop. It depends on two sub-frameworks, GUI-Cmds and GUI-Alerts, as shown in Figure 1, and these will also be described here. All three are complete rewrites of some corresponding Java/Java-Swing libraries known as JUICe[2], in Scala.

[1] `http://lafros.com/gui`

[2] acronym for Java User-Interface Client - see `http://lafros.com/juice`

This paper was first presented as a tech talk at *Scala Days 2010* on April 16 at EPFL, Lausanne.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright © 2010 Latterfrosken Software Development Limited

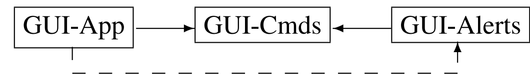

Figure 1. GUI-App depends on two sub-frameworks. The dashed arrow indicates that the GUI-Alerts `.jar` file need only be included in the classpath if it is required by the app itself.

After first giving a brief overview of how the software came to be written, the main facilities which it provides are then presented, using example code to illustrate. The software is then compared with the Java version from which it was derived, and finally, some conclusions drawn.

2. Development history

This software originates from a C++/Motif[3] application framework, MotifApp [1], written[4] as part of the user-level monitoring and control module [2] for a radar to be used for studying the Earth's ionosphere. This was based on the similarly named framework described by Young [3, Chapter 6], and adapted for use with a GUI builder (X Designer).

MotifApp was later partially rewritten[4] in Java as a feasibility study [4], and the result used as the starting point for the experimental Java monitoring and control software that was developed[4] for the same radar subsequently.

Development of this experimental software was later continued independently[4], resulting in JMaCS [5] and JUICe. Note that the former has already been rewritten in Scala, as Lafros MaCS[5] [6].

3. Deployment as application or applet

The main purpose of GUI-App itself is to provide a way to write an app so that the same code may be deployed either as an application (perhaps using Java Web Start[6]) or as an applet (embedded in a browser window). All that is

[3] X Window toolkit

[4] by the author

[5] `http://lafros.com/macs`

[6] `http://java.sun.com/developer/technicalArticles/WebServices/JWS_2/JWS_White_Paper.pdf`

required is that the app should include a singleton object, conventionally called app, that extends ~.gui.app.App[7], and implements the abstract init method:

```
object app extends App {
  def init(context: Context) {...}
}
```

This may then be run as an application—it inherits the requisite main method—or else its (fully-qualified) name may be given as the value of a parameter named App, of the supplied ~.gui.app.Applet:

```
<applet code="com.lafros.gui.app.Applet"
  ...
  <param name="App" value="org.myorg.myapp.app">
</applet>
```

app inherits suitable default implementations of all the other methods the framework uses: displayApplication, start, stopApplet, restartApplet, terminate. All except main will be called from a java.awt.EventQueue dispatch thread.

4. GUI commands

4.1 Keeping the user informed

GUI-Cmds provides the ~.gui.cmds.Cmd trait, which is simply an abstract function (i.e., having an abstract apply method) that returns an optional feedback message:

```
val cmd = new Cmd {
  def apply() = {
    ...
    Some("useful thing done")
  }
}
```

The above may also be written,

```
val cmd = Cmd {
  ...
  Some("useful thing done")
}
```

The feedback message will be passed to the succeeded method of any ~.gui.cmds.CmdsApp registered with ~.gui.cmds.TheCmdsController.instance, and displayed accordingly. In the case of a ~.gui.app.App, this will result in its being displayed using the ~.gui.app.MsgLine assigned to context.msgLine by the init method.

4.2 Provision of common functionality

Various Cmd subtraits are also provided, to support commonly required variations in the method of execution. Thus, a CheckFirstCmd prompts the user for confirmation, a PwdProtectedCmd prompts for a password, and a SeqBgCmd is executed in the background. These may be extended in any combination.

4.3 Robust environment

Any exception thrown by a Cmd's apply method[8] will be caught, and passed to the failed method of the CmdsApp instance mentioned above. Once again, this will result in a message being displayed on a ~.gui.app.App's message line, this time optionally[9] accompanied by an audible alert.

Where input must be validated, or where the effect of the user interaction is generally less predictable (as in control applications, especially over a network), having such a built-in system for recovering from and reporting exceptional conditions was found to be highly desirable.

4.4 Toggles

One further Cmd subtrait (that may also be used in combination with the others) is Tog.Cmd, which is for use where a command represents a toggle. Implementations must supply a ~.gui.cmds.Tog instance, whose state changes after—and *only* after—the command has been executed successfully, i.e., after its apply method returns.

It is also possible to represent a toggle by combining two non-Tog.Cmd commands, as shown in the following section.

4.5 Exer, Trig and Trig.Props

To benefit from the facilities described above, the commands must be executed indirectly, via an appropriate ~.gui.cmds.Exer, which may be instantiated and invoked as follows:

```
val exer: Exer = Exer(cmd)
val togExer1: Tog.Exer = Exer(togCmd)
val togExer2: Tog.Exer = Exer(setCmd, resetCmd)
exer.executeCmd()
togExer1.executeCmd()
...
```

Note that, in contrast to a Cmd's apply method, executeCmd is guaranteed to return, immediately.

Rather than have to call executeCmd explicitly from one of a scala.swing.AbstractButton's reactions, a mix-in, ~.gui.cmds.Trig, is provided, that allows a similar end to be achieved simply by setting the acquired cmd or exer property:

```
val but = new Button with Trig {
  cmd = myCmd
}
```

The above automatically creates a corresponding Exer, and assigns it to exer. Conversely, assigning a value to exer will assign the corresponding Cmd to cmd.

[7] where ~ denotes com.lafros

[8] or by any other method of any of the Cmd subtraits provided

[9] user-configurable

There is also a `cmdReaction` property, that combines command-like execution with `scala.swing.Reactions.-Reaction`-like event-matching:

```
val but = new Button with Trig {
  cmdReaction = {
    case ActionEvent(_) =>
      ...
    Some("useful thing done")
  }
}
```

The above automatically creates a corresponding `Cmd`, and assigns it to `cmd` (which in turn sets the `exer` property).

Mixing-in `Trig` also allows the `AbstractButton`'s `text` property to be determined by the `Exer` (via the `exerToText` property, of type `Exer => String`), including updating it to reflect the state of the `tog` in the case of a `Tog.Exer`. Note that a `Trig` mix-in is always required when wishing to invoke a `Tog.Exer` using an `AbstractButton` which indicates its `selected` state graphically, in order that this should remain synchronised with the state of its `tog`.

Considering that `scala.swing.Actions` are somewhat heavyweight, it was decided that their use with `Exers` (to allow the properties of all associated `Trigs` to be set in one place) should be optional.

Therefore, a concrete subclass of `Action`, called `~.gui.cmds.Trig.Props`, was introduced, whose `apply` method does nothing. Thus, when required, an instance may be configured appropriately, and made available along side the corresponding `Exer`, in order that it may be set as the `Trig`'s `action` property (when setting its `exer` one). Note that it will be treated as a special case, since setting the `action` property would otherwise displace the `exer` one.

5. Monitor alerts

5.1 `MonField`

GUI-Alerts is based on a `scala.swing.Label` that has been specialised for displaying status values, to be updated periodically. Thus, the `~.gui.alerts.MonField` has an `alert` property, having the following possible values:

NoAlert normal background colour

NonIntrusive red background

Intrusive alternating background colour, accompanied by alert sound

Acknowledged red background.

This may be set either explicitly, or via the `valueToAlert` property (of type `Any => Alert`), which is used whenever the `value` property (of type `Any`) is set, which in turn sets the `text` to `value.toString`.

There is also a property, `templateText`, which, when set to anything other than `""`, prevents the label from being resized whenever its `text` is set; this is a desirable thing to do when many fields are being updated.

5.2 Responding to alerts

Given that monitor windows may have large numbers of fields, which may only be of interest when an alert is raised, a container, `~.gui.alerts.TogPanel`, is provided, that allows them to be hidden away (by deselecting a check box), in the manner of a folding editor. This will then open automatically when any of its `MonField` children (which must be registered, using `listenTo`) raise an alert, and then close again afterwards.

The app as a whole may be notified when any alerts are raised, via a mechanism corresponding to that used in GUI-Cmds (involving a `~.gui.alerts.AlertsApp` and `~.gui.alerts.TheAlertsController.instance`). In the case of a `~.gui.app.App`, this will result in an iconised application being de-iconised.

6. Comparison with JUICe

6.1 Usage

Scala's support for properties, as fully utilised by the Scala-Swing API, makes possible a more declarative style of programming, which is well suited to the task of defining GUIs:

```
val a = new A {
  b = new B {
    c = new C {
      ...
    }
  }
}
```

This may be contrasted with the more precedural style afforded by Java's accessor methods:

```
C c = new C();
B b = new B();
b.setC(c);
A a = new A();
a.setB(b);
```

Since the GUI-App API also fully utilises properties, the resulting Scala code is much clearer and concise than the corresponding Java code written to the JUICe.app one.

Using Scala traits in GUI-Cmd, as opposed to the Java interfaces of JUICe.cmds, allows default method implementations to be supplied, which simplifies the task of extending the various `Cmd` subtraits. Also, Scala's `Option` provides a more satisfactory alternative to returning `null` when a `Cmd` is to have no feedback message.

Finally, the app needs no longer supply the `main` method (required in order for it to be run as an application), since Scala makes it possible to arrange for a suitable one to be inherited.

6.2 Framework implementation

Rather than have to provide a corresponding class for each `AbstractButton` that may be used to trigger command execution, as was required in JUICe.cmds, GUI-Cmds needed only provide the single `AbstractButton` mix-in, `Trig`.

Scala's Actors were used for implementing support for `SeqBgCmd`, which requires the use of a background thread. No `synchronized` blocks were required, resulting in code which is straight forward to reason about, as compared with the corresponding code in JUICe.cmds. Note that this support no longer includes a built-in means of interrupting the background thread, which could not always be relied upon; therefore, it is now up to the `SeqBgCmd` implementation to ensure that it does not block indefinitely.

6.3 Deployment

The price to pay for the above advantages is a dependency on the Scala and Scala-Swing `.jar` files. This presents a problem for GUI apps, since they are likely to be downloaded over the network, and these files may well be significantly larger than the app itself. The solution adopted has been to extract from the above dependencies only those `.class` files which are actually needed by the app (and add them to the `.jar` file containing `app`). This may be achieved with the help of a utility such as ProGuard[10], and a suitable Maven plug-in[11] was written.

7. Conclusion

The Lafros GUI-App application framework and associated sub-frameworks have been presented, and it has been shown how they may be used to simplify writing monitoring and control-oriented user interfaces in Scala. They have also been compared with their Java/Java-Swing predecessors, where it was shown how Scala's support for properties, as fully utilised by both Scala-Swing and GUI-App itself, allows code to be written in a more declarative style, that is much clearer and concise than was possible before.

On the other hand, using GUI-App encumbers the app with dependencies on the sizable Scala and Scala-Swing `.jar` files. This may be addressed by extracting from them only those classes which are actually required.

References

[1] R. Dickens. *MotifApp Guide*, Documentation prepared for Radio and Space Plasma Group, Dept. of Physics and Astronomy, Leicester University (Nov, 1997).

[2] R. Dickens. *The UK User Monitoring and Control (UMC) module for the ESR*, Documentation prepared for Radio and Space Plasma Group, Dept. of Physics and Astronomy, Leicester University (Nov, 1997).

[3] D.A. Young. *Object-Oriented Programming with C++ and OSF/Motif*, 2nd ed., Prentice Hall (1995).

[10] `http://proguard.sourceforge.net/`

[11] `http://lafros.com/maven/plugins/proguard`

[4] R. Dickens. *On the advisability of adopting Java for development in the future*, Report prepared for EISCAT group, RAL, UK (Apr, 1998).

[5] R. Dickens. *JMaCS: a Java monitoring and control system*, Proc. of SPIE Vol. 7019, 7019W (2008).

[6] R. Dickens. *Lafros MaCS: an experimental Scala monitoring and control API*, 14th International EISCAT Workshop poster (Aug, 2009).