Cross-platform Desktop Application Development with JRuby and Swing

Originally published in 2007.

Ruby for the desktop

The Ruby programming language is currently best known for building Web applications, primarily with the Ruby on Rails framework. However, Ruby is more than capable for writing graphical desktop applications as well.

The standard Ruby distribution includes code for bindings for Tk, an open-source, cross-platform set of widgets that allows you to create graphical desktop applications. This can be extremely handy, but when installing Ruby from source code you need to be sure you also have the Tk dependencies and make sure the compilation settings include Tk. Further, if you are using Ruby on Windows installed using the excellent "one-click" installer package, you still have to take extra steps to have Tk working, since it no longer supports automatic installation.

Even with Tk set up for Ruby it is somewhat clunky. Tk applications often look like Tk applications; depending on the target platform it can look somewhat ugly. Plus, attempting to create complex interfaces is daunting. Tk is best used for smaller GUI needs.

Available Toolkits

The weakness of Tk has prompted the development of other GUI toolkit options for Ruby. Here are the some of the notable choices:

FxRuby

FxRuby is a Ruby binding for Fox, a GUI toolkit written in C++. It is available for installation using rubygems. There is a binary gem available for Windows; the gem on other platforms will require you to compile native code.

WxRuby

WxRuby is a binding for the cross-platform wxWidgets C++ GUI toolkit that allows the creation of native-looking desktop applications. It is available for installation as gem.

QtRuby

QtRuby gives you Ruby bindings to the Qt toolkit (the one used in the KDE desktop system). There is a gem for the Windows installation, but only source code for other platforms.

GTK-Ruby

GTK is the UI toolkit used in GNOME; you'll need to compile native code to get this running.

Shoes

Shoes is a recent entry into the Ruby GUI widget world. Unlike the previously mentioned toolkits it is designed specially for Ruby. It is available for installation using platform-specific installers.

Swing

Swing? Yes, Swing, the GUI library that is bundled with every installation of a Java runtime environment. If you run JRuby, then you can use Swing.

All but one of these are GUI or widget libraries written in C or C++, with bindings that allow them to be called from other languages, such as Ruby, Python, and Perl. In almost all cases you'll have to face a number of considerations, such as installation, distribution, and such.

Considerations for selecting a GUI library

What GUI tool set you use will, of course, depend on your particular needs. Here are some criteria to consider:

- A rich set of widgets or components
- Solid implementation
- Availability on multiple platforms (mostly Mac, Win32, KDE, and Gnome)
- Native look-and-feel for hosting platform
- Actively maintained
- Ease of creating custom widgets
- Non-restrictive license
- Affordable cost
- Existing frameworks and libraries to speed development
- Mature IDEs and form layout tools
- Testing tools and frameworks
- Ease of packaging and deployment

If all you want is to toss up the occasional message box, or ask a user for some simple input, almost any of the mentioned toolkits will do. For simple requirements you are likely best to focus on platform availability, a suitable range of widgets, and appropriate cost. If you plan on distributing your application you'll want to check the toolkit licensing. You also must be sure that either the end-user will already have the required environment or that you can easily bundle up all the needed libraries and widgets in either a standalone application or a installation package.

Once you move to more complex applications, though, the requirements get tougher. For any application that goes beyond a few simple forms you almost certainly want to have a form designer tool. You'll also want a rich set of available widgets; you're usually better off reusing, for example, an existing date picker or file browser component than writing your own.

While the various C-based Ruby GUI toolkits each have there share of good qualities, none of them has emerged as a clear winner. They offer no obvious choice for general Ruby cross-platform desktop development.

They each, to varying degrees, have issues with installation, documentation, design tools, packaging, and deployment. Notably, none of these can beat, feature-for-feature, the one non-C option.

Leveraging an existing technology: Java

JRuby is an implementation of Ruby for the Java platform. It allows you to execute Ruby code through the JVM. Ruby code running under JRuby can also load and use Java libraries, and that includes Swing.

Whatever your feelings about Java the language (often a contentious topic among Ruby programmers), it's hard to deny certain aspects of Java the platform:

- Solid and well-tested
- Strong community and vendor support
- Good, plentiful documentation
- Excellent choice of IDEs and UI layout tools
- Free to use (both in cost and license)
- Probably already installed on end-user's machines

If you write an application in (J)Ruby and use Swing for the UI, you need only ensure that the end user has a recent version of the Java runtime, and package your application to include the JRuby jar. And there's a Ruby tool for JRuby application packaging, making that part a non-issue.

Some arguments against JRuby + Swing

On the bright side, there are boundless Swing components; on the dark side, they are *Swing* objects; while you can load these into your Ruby code, you will need to know their assorted APIs and syntactical quirks. There is copious Swing documentation and examples, but unless and until people write Ruby wrappers around then there is no getting around having to get your hands a bit dirty with some raw Swing code.

A second common objection is that Swing may not look quite as slick or native as, say, Qt. However, things are not what they used to be, and Swing can now look quite good.

Weighing the a Java platform against a C-based tool, you'll have to make some choices and decide what's most important. For example, Swing is completely free to use, while Qt has various costs sand restrictions for commercial and open-source applications. On the other hand, the look-and-feel of the Qt components may be better for your program.

Developer options for (J)Ruby + Swing

There are a number of options for using Swing from Ruby

Raw, hand-coded, in-line calls to Swing objects

In the simplest case, you can refer to Swing objects much as you would any other Ruby object:

```
panel = Java::javax::swing::JFrame.new("JRuby panel")
panel.show
```

"Builder" and DSL-style libraries

Constructing panels and forms then adding component all in hand-rolled code can get tricky quite fast. Some libraries exist to make the Swing interaction more Ruby-like. For example, Cheri::Swing uses Ruby block syntax to generate the Swing code.

Another library, Profligacy, offers a Ruby wrapper around the raw Swing calls to help you write more Swing code with less raw Java code. You'll still need to get familiar with th Swing API docs to make proper use of the Swing components.

Both of these approaches presume the creation of panels and forms and layouts using hand-crafted code. While an improvement on doing this with straight-up Swing code, they still suffer from an inability to handle complex user interfaces.

The "We don't care where the Java class came from" approach

A third approach is to simply punt on trying to ease the creation of Swing objects using Ruby code, and start from the assumption that a compiled Java class for the Swing objects already exists.

This is the approach taken by the Monkeybars library. There are a number of very good, free, graphical Swing UI layout editors. As with the use of the previously mentioned GUI toolkits (e.g., Fox, GTK), you don't need a UI editor for the occasional dialog box. But it's hard to beat such a tool for anything more than that, and it's pointless to skip these tools and code the UI by hand for a sophisticated desktop application. So go for it.

Monkeybars

Monkeybars is a Ruby library that connects existing Java Swing classes (that is, compiled Java classes that define your Swing UI) to Ruby code using a form of MVC. MVC , or model, view, controller, is a design pattern aimed at separating view logic and user interface components from application logic. Monkeybars is an open-source project that grew out of product development at Happy Camper Studios. It is actively maintained and completely free to use as you see fit, and has evolved based on experience trying to create testable, maintainable complex Ruby desktop applications.

Since it uses the Java language and Swing libraries it is building on mature, robust technology. Unlike other current Swing libraries for JRuby it is well-suited for constructing large, complex, multi-paneled applications. As we'll see, there is some overhead in creating a Monkeybars application, so it may not be your best choice for simple forms. It is, however, a reasonable choice for a JRuby desktop application of any complexity when you need:

- Reliable cross-platform deployment (assured to the degree that the end user has a recent JVM installed)
- A large choice of UI widgets of arbitrary complexity
- Possibly complex UI form/panel construction and interaction

As with Profligacy, Monkeybars does not hide the Swing API. Nonetheless, because it works with compiled UI classes, you can make full use of any tool or application to generate the actual layout. Depending on the complexity of your application it is almost inevitable that at some point you will need to reference Swing component API documentation and code examples to know what to do in your Ruby code. (But because of the nice integration of JRuby and Java libraries you can easily wrap such Swing code in a Ruby API for easier reuse.)

An example JRuby Swing application using Monkeybars

To get a feeling for building for creating a desktop application with Swing and Ruby, we'll take a tour through a simple program created using Monkeybars.

Installation

To get started you'll need to have a few things in place. First, you'll need a copy of the `jruby-complete.jar`. The simplest way is to download the file from

Why Swing instead of SWT?

- Swing is a part of Java; if a user has Java, they have Swing.
- Swing allows for more fine-grain control of component behavior
- Swing components offer more flexibility over how components may look

`http://dist.codehaus.org/jruby/`.

Next, you need to install the Monkeybars library. You can install it as a gem:

```
sudo gem install monkeybars
```

You can also grab the current source code from the repository on gitorious.org (`http://gitorious.org/projects/monkeybars`).

Finally, you need to install the `rawr` gem. Strictly speaking , it is not required when writing a Monkeybars application, but it provides a number of useful Rake tasks for turning a JRuby application into an executable jar file and this example will use it.

```
sudo gem install rawr
```

Application basics

Programs built using Monkeybars can be arbitrarily complex, but there are some basic patterns to follow to keep the code manageable. The example application will be a “flash card” program: it will read in a text file that defines a number of “cards”; it will loop until shut down, periodically showing and hiding itself for brief periods. Basically, it's a tool, for learning . For this example the cards will be a set of German vocabulary words and phrases. The program will also read a configuration file that defines the location of cards definition file and a few settings (show/hide speed, window size).

The goals of this example are to:

- Show the use of Monkeybars code generators, which automate the creation of common files
- Show the basic structure of a Monkeybars application
- Demonstrate the creation of each part of the Monkeybars MVC tuple
- Show how Monkeybars handles the mapping of application data to UI components
- Show packaging the application as an executable jar file

The Monkeybars approach to Model View Controller

The MVC pattern has a long history, with numerous variations. Here's how it works in Monkeybars:

The basic presumption is that for each Swing frame (that is, the UI object holding assorted components or widgets; in some cases this may be modal panel) there are three Ruby files: a model, a view, and a controller. The model class holds the essential business logic and manages the data corresponding to that part of the application. It should be ignorant of any view or controller code which exist as a means to interact with the model. Keeping view and controller references out of your model makes it much easier to develop and test.

The view is another Ruby file with a reference to a specific Java class containing the compiled Swing code. The view manages the interaction of Swing components with model data. While the view may have direct contact with the model, it also works with a copies of the model as mens for passing data to the controller. This is important to keep in mind when designing your model class, since it ends up serving a dual purpose. The primary instance of the model is keeping long-term state and providing application logic; the copy used by the view is essentially a disposable data access object. Models should be relatively cheap to instantiate, with shallow accessors provided for any data used by the view.

The view does not have direct contact with the controller. Instead, there is a signaling system in place to abstract the interaction of controller and view. This decoupling makes it easier to test your views and controllers.

(This description is deliberately simplified and omits various details. Under the hood there is closer interaction between view and controller; the infrastructure needs the means to coordinate behavior. The goal of Monkeybars is not to tie your hands but to

assist you in creating testable, maintainable code. As a developer, though, you are free to bypass the intended API if you see fit.)

The controller class is where you define handlers for Swing events (such as button clicks and text field changes) and control the state of into the model. The controller keeps a reference to the primary instance of the model. It does not communicate directly with the view.

When a controller wants to get data from the view, the view provides a copy of the model populated with the current UI contents. The controller can then decide to update the primary instance of the model with this data, or take some action based on these values. The controller can also tell the view to update itself and pass back updated values. We'll see this in action in the example.

Using the Monkeybars application generator script

Once installed, Monkeybars provides a command line script to create an initial set of application files. To start a new Monkeybars project you should execute the `monkeybars` script that is installed with the gem. We'll name our project `monkey_see`:

```
$ monkeybars monkey_see
```

This will create a new directory at the given path (or in the current directory if you only give an application name) and add core files and directories for the new application.

Using Rawr to bootstrap the code into the Java environment

Rawr is another Ruby library that grew out of Monkeybars. It handles assorted packaging tasks, and provides a command line script for creating a base Java class that a Monkeybars application can use to allow execution as a Java program (as opposed to running the application as a Ruby program via JRuby).

You use it with your Monkeybars application by going to your project directory and executing the `rawr` script:

```
$ cd money_see; rawr install
```

Using Monkeybars Rake tasks to generate files

We've seen how Monkeybars splits things up into model, view, and controller. The convention is to place these files into the same directory. The help this along, Monkeybars provides a Rake task to generate these files.

You can create one of the three, or a full set (the more common case):

```
$ rake generate ALL=src/flash
```

This will create a subdirectory `flash` under `src/`, with three files: `flash_controller.rb`, `flash_view.rb`, and `flash_model.rb`. The first

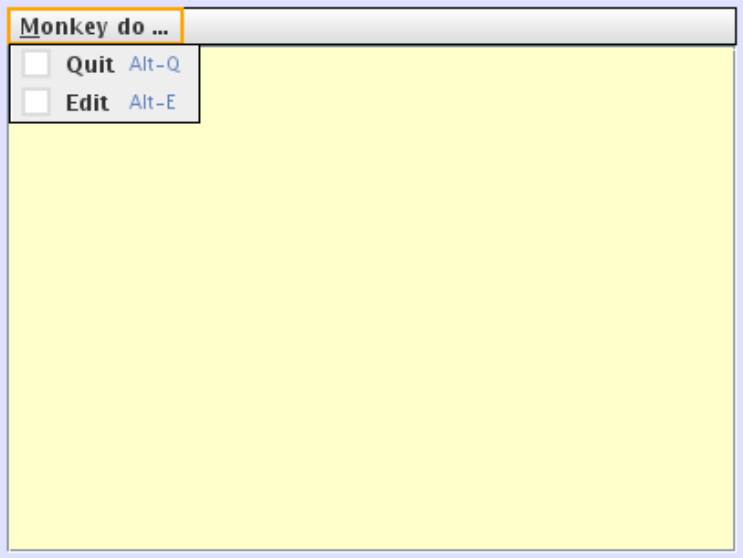
two will have bare-bones classes that inherit from base Monkeybars classes. The model code does not; Monkeybars makes zero assumptions about how you manage your application logic and data; that is entirely up to you.

Creating the UI

For the application's interface we need a Swing class that will display the flash card data.

How you create this this is up to you; there's nothing in Monkeybars that ties to any particular UI tool or Swing code generator. By convention, the Swing files get placed in the same directory as it's related tuple (I.e. `src/flash/FlashFrame.java`). You'll need to know the class package so we can pass it on to the view class. (We'll be using the package `flash` and name the class `FlashFrame`.)

Our screen layout should look like this:



A few key points: We want to use a `JTextPane` for the flash card content so we can use HTML to format the rendered text. We also want to use sensible names for the text pane and the button. It just makes it easier when you're working with the view to know something about the UI components. Since the program code is in Ruby , use Ruby method name conventions; call the text `pane card_pane`, and the two menu items `edit_menu_item` and `quit_menu_item`. Give than accelerator keys, too.

The name of the frame itself is not important; the view class can reference the components directly by name.

Defining the model

The model manages the application logic and data behind a given UI. A Monkeybars program will generally have a model for each Java form. Our application has but one model to handle the flash card data. The model code needs to be able to load data from a known location, and offer a public method to provide that data.

For simplicity we'll store the data in a text file in a subdirectory from where the application is running . Rather than hand-code HTML, we can use Textile markup and transform it using the RedCloth Ruby library. Each card entry will be separated by a delimiter string.

Using 3rd-party libraries

Textile is a text markup format that is intended to define HTML using simple plain-text conventions. For example, to indicate `<em>italicized</em>`, you instead write `_italicized_`. RedCloth is Ruby library, available as a gem, that converts Textile formated text into HTML.

Rubygems makes it quite easy to install and use 3rd-party libraries, but as we will want to package our code in a jar and potentially distribute it to people we need to be sure that all code is included with the application. To do this, you need to unpack the RedCloth gem and copy the `redcloth.rb` file the project's `ruby/lib/` directory.

```
$ cd /tmp; gem unpack RedCloth
```

This will create `/tmp/RedCloth-3.0.4 /`(unless you have a different version of the gem installed). Copy `/tmp/RedCloth-3.0.4/lib/redcloth.rb` to the `lib/ruby/` directory of your `monkey_see` project.

In general, any Ruby libraries that are not core parts of your application should go (as a convention) under `lib/ruby/`. If you are using gems, you need to unpack the actual library files and add them to your project. Later in this article you'll see how to tell your program how to find these files.

Key model methods

The `load_cards` method will handle reading in the text file from disk, splitting out each card, and assigning the results to the `@cards` instance variable.

A `select_card` method will pick a card at random and assign it the `@current_card` instance variable. We'll use `attr_accessor` to define methods for reading and setting this variable.

We'll arrange it so that whatever card is being displayed in the UI may be edited in-place. After editing, the `update_current_card` method will take the contents of `@current_card` and reinsert it into the `@cards` array. A `save` method will write the `@cards` array back to disk.

The value of the `current_card` method is what we want to render, and to do that, we'll need a view class.

Defining the view class

A Monkeybars view class is the owner of the Java Swing class. If you open up `flash_view.rb` you'll see that it invokes a class method, `set_java_class`. This

should be set to the Swing class defined for this view. In our code, this is `flash.FlashFrame`.

In general, a Monkeybars view class needs to do a three things: pass data in and out of the Swing components; manage assorted view-centric behavior (such as size and position); respond to signals sent from the controller.

Mapping data

Monkeybars provide a `map` method that allows you to define how model methods are wired up to Swing controls. The simplest usage connects a UI component method and a model method:

```
map :view => :card_pane.text, :model => :current_card
```

This mapping will use the default behavior of making this a direct, two-way association. That is, the results of the `text` method of the `card_pane` component will be passed to the `current_card=` method of the model. When updating the view from the model, it is reversed: `model.current_card` will populate `card_pane.text` (note: JRuby handles Ruby/Java naming conversion, so the actual Swing method, `setText`, may be invoked using `set_text =`).

Quite often this form of simple mapping works fine, but there are times when, because of differences in data types or formatting, or due to the needs of some application logic, you don't want direct data exchange. Monkeybars allows the use of intermediates in the data exchange. A mapping can be passed a `:using` parameter (that is, a hash key pointing to an array) that indicates the alternative means to use when moving data from the model to the view, and from the view to the model. (Another reason for `:using` is when the value or state of a Swing component needs to be manipulated using component methods or child objects that do not fall into the general `getProperty` and `setProperty` pattern.)

For our code we want to take a Textile-formated string from the model and convert it to HTML before assigning it to the `card_pane` `text` property. To handle this we'll create a `to_html` method. We also don't want to directly update the model's `current_card` value from the view. We'll have some special code for editing cards in the view, so we'll use `nil` in place of what would otherwise be some view-to-model method name.

That gives us this map:

```
map :view   => :content_pane.text,
    :model => :current_card, :using => [:to_html, nil ]
```

We'll also want our Swing frame to present itself in a specific manner. By default, a Swing frame will appear in the top left corner of the screen. For our application we want it to show in the top right corner. We'll also give it a nice sliding effect so that it does not abruptly come and go.

Managing the Swing object

A view class has a special instance variable `@main_view_component` that references its corresponding Swing class. It is through this object that view code interacts with Swing components. To change the content of the flash card text pane, for example, you could write

```
@main_view_component.card_pane.text = "Some new text"
```

However, since this kind of code is essentially the reason the view class exists, Monkeybars arranges it so that you can omit explicit use of `@main_view_component` and refer directly to its components:

```
card_pane.text = "Some new text"
```

The base `Monkeybars::View` class uses `method_missing` to intercept such code, looks to see if it is a component reference, and if so it delegates the request to `@main_view_component`.

Method calls on the Swing class, though need the explicit reference:

```
#width = 500  # Fails; no such method as 'width'
@main_view_component.width = 500
```

To achieve a nice sliding effect the view class has methods that manipulates the height and position of the Swing frame, gradually expanding and contracting it so that, on each rendering cycle, it slides down from the top of the screen, then slides back up.

Handling requests from the controller

Monkeybars is designed to decouple key parts of the MVC tuple. Since the view has a direct reference to a Java Swing object it is typically the hardest part to test. Monkeybars aims to reduce direct view interaction with the model and controller. The controller, however, is responsible for handling UI events. Inevitably this means the controller needs to direct the view to respond. The controller, though, does not directly communicate with the view class. Instead, it uses *signals*.

We'll see the controller side of this shortly. In the view, you need to define signal handlers using the `define_signal` method. It takes a hash defining a signal name and a view method to handle that signal.

```
define_signal :name => :initialize,
              :handler => :do_roll_up
```

Handler methods must take two arguments: the model (passed in from the controller), and a *transfer* object. The transfer object is a transient hash used to move data back and forth between controller and view. Our view will have signals defined for the initial positioning of the UI, the slide in, slide out sequence, and two for beginning and ending

card editing. Each of these signal handlers is quite short. Here's the `do_roll_up` method:

```
  def do_roll_up model, transfer
    hide
    move_to_top_right
    roll_up
  end
```

The editing sequence will be triggered through menu events. The `Edit` menu item toggles editing. In the view, the editing sequence means setting `card_pane.editable = true`, then swapping out the HTML-rendered content with the raw Textile card text. We also have to change the content-type of the component so it will correctly render plain text.

When editing is complete, the reverse is done. The pane is given HTML, and `editable` set to false. The view is only concerned with managing the state of the Swing component; the controller will handle instructing the model to perform the text updating and saving.

Defining the controller class

Our Swing object has some menu items, but we've not put any code for them in the view class. That belongs in the controller. The controller handles all UI events, such as button clicks, menu selection, and text field changes. Monkeybars arranges it so that by default all such events coming from the Swing code are quietly swallowed. You need to define event handlers for just those things you care about. In our code that would be menu clicks.

Event handlers take this form:

```
    def your_component_name_action_performed
      # code
    end
```

(You may also define the handler to take the actual Swing event as a parameter should you want your code to use it.)

To handle the `Quit` menu item we just need to exit:

```
  def quit_menu_item_action_performed
    java.lang.System.exit(0)
  end
```

The `Edit` menu action needs a bit more:

```
  def edit_menu_item_action_performed
    if @editing
```

```
      @editing = false
      signal :end_edit
      update_card
    else
      @editing = true
      signal :begin_edit
      update_model view_model, :current_card
    end
  end
```

The code handles the toggling of editing modes, using signals to drive the view. The key thing to note is how card text is moved using the controller's model instance (implicitly passed to the view by way of the signal), and the view's copy of the model, provided by the `view_model` method.
Whenever a controller need the current state of the user interface, it can use the `view_state` method to reference the view's copy of the model and the current transfer object. Since grabbing the model copy from `view_state` is so common, Monkeybars provides the the `view_model` method.

Our controller will also have a method to kick off the initial rendering, and another that handles the show/hide display sequence. Both use signals to defer the actual presentation code to the view.

Orchestrating the application

In addition to one or more MVC tuples, a Monkeybars application uses two key helper files to prepare and run your code.

Both are in the `src/` directory. The `manifest.rb` file sets up library load paths, and allows you to define what files are to be included based on whether the program is run straight from the file system or from a jar file.

Earlier we added `redcloth.rb` to `lib/ruby/`. In order for your application to locate this file you need to add that directory to the load path. The same goes for the `lib/java/` directory, So, make sure that `manifest.rb` has these lines:

```
add_to_load_path "../lib/java"
add_to_load_path "../lib/ruby"
```

Also in `src/` is `main.rb`. This is the Ruby entry point for the application. Among other things, it defines a global error handler and is where you would place any platform-specific code to run prior to executing main application logic.

Our program will use a simple loop:

```
begin
  flash_card = FlashController.instance
```

```
  flash_card.init_view :flash_interval_seconds => 8,
                       :show_for_seconds => 20,
                       :window_height => 200,
                       :data_src => 'data/cards.rc'

  while true do
    flash_card.present
  end

rescue => e
  show_error_dialog_and_exit(e)
end
```

Monkeybars controllers are typically used as singleton classes; you generally do not need multiple instances. (A notable exception is the use of *nested controllers* where a frame or panel has multiple subcomponents that are each instances of the same class, a topic beyond the scope of this article. Basically it allows you to take a complex set of frames, panels, and components and break it down into a set of individual MVC tuples. An example might be an address book, were a top-level Swing frame renders multiple address objects, each being an instance of an `address_entry` MVC set.)

Executing the code

With the code in place and a suitable data file we can run the program. We'll use a `rawr` Rake task to create an executable jar file. When we ran `rawr install` at the start of the project it created a `Main.java` file under `src/org/rubyforge/rawr/`. To run the program from a jar, there needs to be a `Main` Java class; `rawr` generates this file containing basic code that looks for and interprets a `main.rb` file. (Or, if one is not found, it invents one in-line and uses that instead.)

The Rake task `rawr:jar` will compile this code and package up our files into a jar. To coordinate this there is the file `build_configuration.yaml`. Prior to creating the jar this should be edited to reflect the details of the application.

To kick off our program, first we build the jar file:

```
$ rake rawr:jar
```

and then we invoke it

```
$ java -jar package/deploy/monkey_see.jar
```

We should see the flash card screen roll down from the top right corner, stay for a bit, then roll back up.

When the window is visible, we can use the menu items to edit the currently displayed card. To quit, we use the `Quit` menu item (`Alt+Q`, if you've added the accelerator key).

Some caveats about the generated jar file

The jar file created by `rawr:jar` does not contain everything needed to execute the program. In particular, any jars containing libraries needed for the program (such as `jruby-complete.jar` and `monkeybars-0.6.4.jar)` are not bundled in `monkey_see.jar`.

When `rawr:jar` runs, it creates a set of files under `package/deploy/`. These are the files needed to execute the application, and must be distributed together. The `build_configuration.yaml` file allows you to steer what files are placed n the jar, and what files and directories are copied over to the deployment directory. The `data/cards.rc` file, for example, *could* be bundled into `monkey_see.jar`, but as the application allows editing it should be left as an external file so that changes may be written back.

Conclusion

The development of JRuby as a robust, viable alternative to the traditional C implementation of Ruby means that Ruby GUI toolkits can move beyond C-based options and use UI tools available to the Java platform. As Swing is a standard part of a Java runtime installation, Swing components give (J)Ruby a mature and readily available graphical toolkit. Employing the Java platform means such applications can be readily built, packaged, and distributed to end users on multiple platforms. When combined with the Monkeybars library, Ruby developers can build testable, maintainable, complex desktop applications with increased ease.

The example here was deliberately small, intended mainly as introduction to what is possible for JRuby Swing GUI development. More information about Monkeybars, with larger examples, may be found at the `http://www.monkeybars.org`

Resources

- API documentation, tutorials, and examples for the Monkeybars library may be fund at `http://www.monkeybars.org`
- JRuby document and downloads are available at `http://jruby.org/`