Technical Explorations

by Keith R. Bennett

Ruby's Forwardable

Sep 13, 2012 • keithrbennett

Last night I had the pleasure of attending the Arlington Ruby User Group meeting in Arlington, Virginia. Marius Pop, a new Rubyist, presented on Ruby's Forwardable module. Forwardable allows you to very succinctly specify that you want to define a method that simply calls (that is, delegates to) a method on one of the object's instance variables, and returns its return value, if there is one. Here is an example file that illustrates this:

```
>require 'forwardable'

class FancyList
  extend Forwardable

  def_delegator :@records, :size

  def initialize
    @records = []
  end

end

puts "FancyList.new.size = #{FancyList.new.size}"
puts "FancyList.new.respond_to?(:size) = #{FancyList.new.respond_to?(:size)}"

# Output is:
# FancyList.new.size = 0
# FancyList.new.respond_to?(:size) = true
```

After the meeting I thought of a class I had been working on recently that would benefit from this. It's the LifeTableModel class in my Life Game Viewer application, a Java Swing app written in JRuby. The LifeTableModel is the model that backs the visual table (in Swing, a *JTable*). Often the table model will contain the logic that provides the data to the table, but in my case, it was more like a thin adapter between the table and other model objects that did the real work.

It turned out that almost half the methods were minimal enough to be replaced with Forwardable calls. The diff is shown here:

```
1  diff --git a/lib/life_game_viewer/view/life_table_model.rb b/lib/life_game_viewer/view/life
2  index 0ee2966..6cbcba1 100644
3  --- a/lib/life_game_viewer/view/life_table_model.rb
4  +++ b/lib/life_game_viewer/view/life_table_model.rb
5  @@ -3,15 +3,26 @@ require 'java'
6   java_import javax.swing.table.AbstractTableModel
7   java_import javax.swing.JOptionPane
8
9  +require 'forwardable'
10 +
11  require_relative 'generations'
12
```

```
 # This class is the model used to drive Swing's JTable.
 # It contains a LifeModel to which it delegates most calls.
 class LifeTableModel < AbstractTableModel

+  extend Forwardable
+
   attr_accessor :life_model
   attr_reader :generations

+  def_delegator :@life_model, :row_count,       :getRowCount
+  def_delegator :@life_model, :column_count,    :getColumnCount
+  def_delegator :@life_model, :number_living
+  def_delegator :@life_model, :alive?,          :getValueAt
+
+  def_delegator :@generations, :at_first_generation?
+  def_delegator :@generations, :at_last_generation?

   def initialize(life_model)
     super()
@@ -24,34 +35,10 @@ class LifeTableModel < AbstractTableModel
     @generations = Generations.new(life_model)
   end

-  def getRowCount
-    life_model.row_count
-  end
-
-  def getColumnCount
-    life_model.column_count
-  end
-
-  def getValueAt(row, col)
-    life_model.alive?(row, col)
-  end
-
   def getColumnName(colnum)
     nil
   end

-  def at_first_generation?
-    generations.at_first_generation?
-  end
-
-  def at_last_generation?
-    generations.at_last_generation?
-  end
-
```

```
60  -  def number_living
61  -    life_model.number_living
62  -  end
63  -
64     def go_to_next_generation
65       if at_last_generation?
66         JOptionPane.show_message_dialog(nil, "Generation ##{generations.current_num} is the
```

gistfile1.diff hosted with ❤ by GitHub view raw

The modified class is viewable on Github here.

As you can see, there was a substantial reduction in code, and that is always a good thing as long as the code is clear. More importantly, though, `def_delegator` is much more expressive than the equivalent standard method definition. It's much more precise because it says this function delegates to another class' method *exactly*, in no way modifying the behavior or return value of that other function. In a standard method definition you'd have to inspect its body to determine that. That might seem trivial when you're considering one method, but when there are several it makes a big difference.

One might ask why not to use inheritance for this, but that would be impossible because:

a) the class delegates to three different objects, and

b) the class already inherits from AbstractTableModel, which provides some default Swing table model functionality.

Marius showed another approach that delegates to the other object in the method_missing function. This would also work, but has the following issues:

a) It determines whether or not the delegate object can handle the message by calling its *respond_to* method. If that delegate intended to handle the message in its method_missing function, respond_to will return false and the caller will not call it, calling its superclass' method_missing instead.

b) The delegating object will itself not contain the method. (Maybe the method_missing handling adds a function to the class, but even if it does, that function will not be present when the class is first loaded.) So it too will return a misleading false if respond_to is called on it.

c) In addition to not communicating its capabilities to objects of other classes, it does not communicate to the human reader what methods are available on the class. One has to look at the class definition of the delegate object, and given Ruby's duck typing, that may be difficult to find. It could even be impossible if users of your code are passing in their own custom objects. This may not be problematic, but it's something to consider. (I talk more about duck typing's occasional challenges in another article, Design by Contract, Ruby Style.

It was an interesting subject. Thank you Marius!

Published with GitHub Pages