CHAPTER 1

Flag and country exercises

In this tutorial, we will take a concrete approach to teach you some Pharo code.

- You will learn how to draw a country shape using Roassal (a visualization engine).
- In a second step, you will use an HTTP client to grab the flag of the country based on its unique international id.
- You will then define a small visual application using the Spec UI builder and it will display the country, its ID, and flag.

1.1 Resources

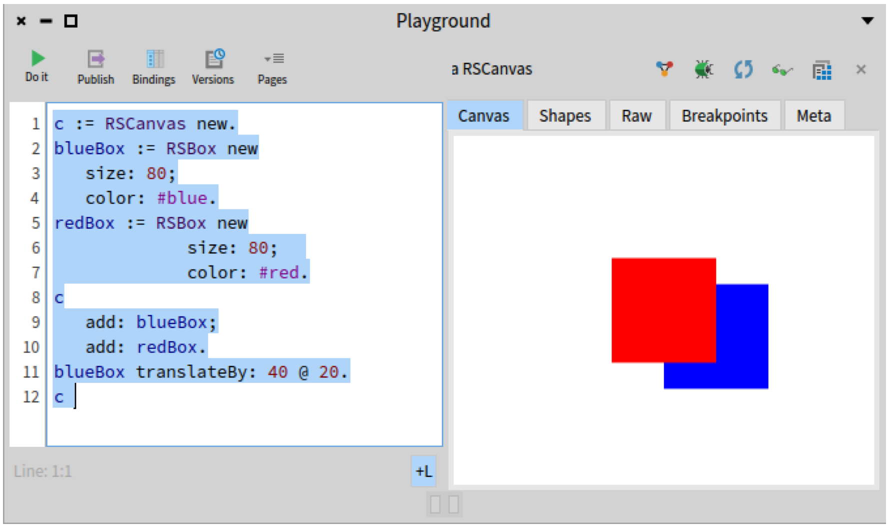
This tutorial uses the following resources:

- The Roassal visualization engine. Roassal - (See https://github.com/pharo-graphics/Roassal).
- An XML parser XML Parser - (See https://github.com/pharo-contributions/XML-XMLParser).
- A file `world.svg`. This SVG file defines country shapes as shown in Figure 1-2. You can find this file at https://github.com/SquareBracketAssociates/booklet-CountryTutorial/.
- The Spec UI builder that supports the definition of user interface. Spec book - (see https://github.com/SquareBracketAssociates/BuildingApplicationWithSpec2/).

1.2 Roassal first script

Let us get started. Inspect the following code snippet to get two boxes drawn in the canvas. You should get an inspector as shown in Figure 1-1.

```
| c |
c := RSCanvas new.
blueBox := RSBox new
  size: 80;
  color: #blue.
redBox := RSBox new
  size: 80;
  color: #red.
c
  add: blueBox;
  add: redBox.
blueBox translateBy: 40 @ 20.
c
```



```
c := RSCanvas new.
blueBox := RSBox new
   size: 80;
   color: #blue.
redBox := RSBox new
            size: 80;
            color: #red.
c
   add: blueBox;
   add: redBox.
blueBox translateBy: 40 @ 20.
c
```

Figure 1-1 A program and its graphical rendering: Two boxes.

1.3 World map

Now we are ready to display countries. The idea is that we will copy the SVG path definition from the world.svg file into a little Roassal program that renders SVG paths.

- Check file world.svg, open it with a text editor.
- Copy the path of a country that you want to display, pick a little country that will help you.

Figure 1-2 World.svg rendered.

1.4 SVG shapes

Now we will display the SVG path you selected. In the following snippet, we put

Inspect the following code snippet: it produces a little bezier surface.

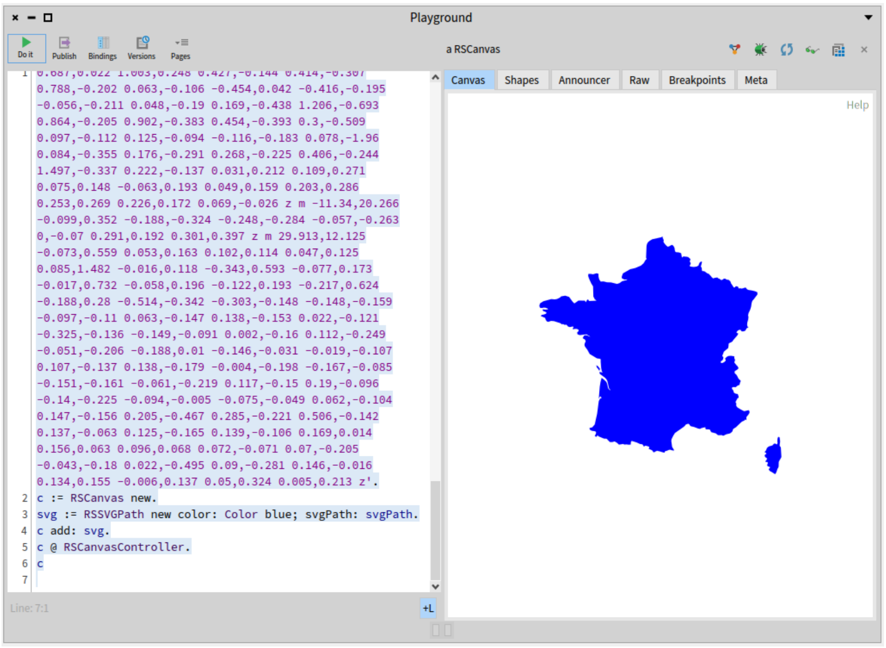
```
| c svg |
c := RSCanvas new.
svg := (RSSVGPath new
  svgPath: 'M 100 350 q 150 -300 300 0';
  yourself).
c addShape: svg.
c @ RSCanvasController.
c
```

Zoom out pressing the key O on the window. You should obtain a not-really exciting graphical form.

1.5 Display a country

Using the path defined in the world.svg file taking, for example, the path of France you should get Figure 1-3.

```
svgPath := 'm 482.92875,298.0884 ...'.
c := RSCanvas new.
svg := RSSVGPath new
  color: Color blue;
  svgPath: svgPath.
c add: svg.
c @ RSCanvasController.
c
```



```
0.687,0.022 1.003,0.248 0.427,-0.144 0.414,-0.307
0.788,-0.202 0.063,-0.106 -0.454,0.042 -0.416,-0.195
-0.056,-0.211 0.048,-0.19 0.169,-0.438 1.206,-0.693
0.864,-0.205 0.902,-0.383 0.454,-0.393 0.3,-0.509
0.097,-0.112 0.125,-0.094 -0.116,-0.183 0.078,-1.96
0.084,-0.355 0.176,-0.291 0.268,-0.225 0.406,-0.244
1.497,-0.337 0.222,-0.137 0.031,0.212 0.109,0.271
0.075,0.148 -0.063,0.193 0.049,0.159 0.203,0.286
0.253,0.269 0.226,0.172 0.069,-0.026 z m -11.34,20.266
-0.099,0.352 -0.188,-0.324 -0.248,-0.284 -0.057,-0.263
0,-0.07 0.291,0.192 0.301,0.397 z m 29.913,12.125
-0.073,0.559 0.053,0.163 0.102,0.114 0.047,0.125
0.085,1.482 -0.016,0.118 -0.343,0.593 -0.077,0.173
-0.017,0.732 -0.058,0.196 -0.122,0.193 -0.217,0.624
-0.188,0.28 -0.514,-0.342 -0.303,-0.148 -0.148,-0.159
-0.097,-0.11 0.063,-0.147 0.138,-0.153 0.022,-0.121
-0.325,-0.136 -0.149,-0.091 0.002,-0.16 0.112,-0.249
-0.051,-0.206 -0.188,0.01 -0.146,-0.031 -0.019,-0.107
0.107,-0.137 0.138,-0.179 -0.004,-0.198 -0.167,-0.085
-0.151,-0.161 -0.061,-0.219 0.117,-0.15 0.19,-0.096
-0.14,-0.225 -0.094,-0.005 -0.075,-0.049 0.062,-0.104
0.147,-0.156 0.205,-0.467 0.285,-0.221 0.506,-0.142
0.137,-0.063 0.125,-0.165 0.139,-0.106 0.169,0.014
0.156,0.063 0.096,0.068 0.072,-0.071 0.07,-0.205
-0.043,-0.18 0.022,-0.495 0.09,-0.281 0.146,-0.016
0.134,0.155 -0.006,0.137 0.05,0.324 0.005,0.213 z'.
c := RSCanvas new.
svg := RSSVGPath new color: Color blue; svgPath: svgPath.
c add: svg.
c @ RSCanvasController.
c
```

Figure 1-3 Displaying France.

1.6 Loading an XML Parser

So far it was fun but too manual. We will use an XML parser. In addition, we will represent each country as an object that we can manipulate later.

Check if the XMLParser is loaded in your image. Else you can load the XML parser available at `https://github.com/pharo-contributions/XML-XMLParser`.

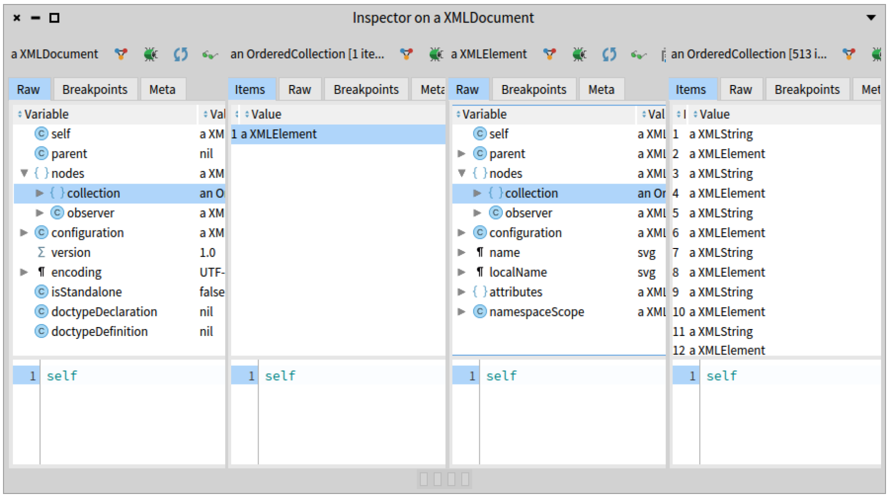
```
Metacello new
  baseline: 'XMLParser';
  repository: 'github://pharo-contributions/XML-XMLParser/src';
  load.
```

1.7 Tweaking XML tree

Using the following snippet, inspect the tree returned by the parser.

```
(XMLDOMParser parse: 'world.svg' asFileReference readStream contents)
    document inspect
```

Dabble the data and find the list of countries. You can see this in Figure 1-4. It shows that the field `'nodes'` contains another collection that contains one collection that finally contains a list of elements.



a XMLDocument — Raw | Breakpoints | Meta

Variable	Val
self	a XM
parent	nil
▼ nodes	a XM
▶ collection	an O
▶ observer	a XM
▶ configuration	a XM
version	1.0
▶ encoding	UTF-
isStandalone	false
doctypeDeclaration	nil
doctypeDefinition	nil

```
self
```

an OrderedCollection [1 ite... — Items | Raw | Breakpoints | Meta

	Value
1	a XMLElement

```
self
```

a XMLElement — Raw | Breakpoints | Meta

Variable	Val
self	a XML
▶ parent	a XML
▼ nodes	a XML
▶ collection	an Or
▶ observer	a XML
▶ configuration	a XML
▶ name	svg
▶ localName	svg
▶ attributes	a XML
▶ namespaceScope	a XML

```
self
```

an OrderedCollection [513 i... — Items | Raw | Breakpoints | Met

I	Value
1	a XMLString
2	a XMLElement
3	a XMLString
4	a XMLElement
5	a XMLString
6	a XMLElement
7	a XMLString
8	a XMLElement
9	a XMLString
10	a XMLElement
11	a XMLString
12	a XMLElement

```
self
```

Figure 1-4 Dabbling the XML node of the files. The inspector lets you navigate a complex structure

Now we would like to convert all the elements into a little class representing countries so that we can manipulate it later.

1.8 The country class

Create a country class.

```
Object << #EarthMapCountry
  slots: { #svgPath . #name . #code };
  package: 'SummerSchool'
```

1.9 Define methods

- Define the corresponding accessors.
- Define the method named `fromXML:` that creates an instance of `EarthMapCountry` from an XML element.

```
EarthMapCountry >> fromXML: aXMLElement

  svgPath := aXMLElement attributeAt: 'd'.
  name := aXMLElement attributeAt: 'title'.
  code := aXMLElement attributeAt: 'id'.
```

- Define the method, `asRSShape` that returns the Roassal SVG shape of a country.

```
EarthMapCountry >> asRSShape
  ^ RSSVGPath new svgPath: svgPath
```

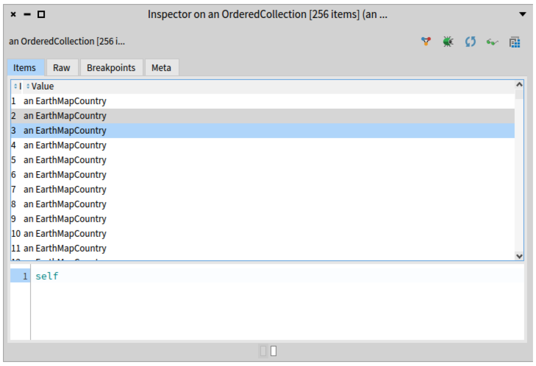
1.10 Grabbing a first country

Now we can convert an element from the XML file and create the correspondingEarthMapCountry instance. (For the XML expert we should not do it that way but use a SAXParser but this is not the point of this tutorial).

```
country := EarthMapCountry new
   fromXML: (XMLDOMParser parse: 'world.svg' asFileReference
     readStream contents) document nodes first nodes second.
country asRSShape inspect
```

All countries

Now let us grab all the countries. The following snippet is a cheap and hacky way to create all the countries from the SVG file.


Inspector on an OrderedCollection [256 items] (an ...

an OrderedCollection [256 i...

I	Value
1	an EarthMapCountry
2	an EarthMapCountry
3	an EarthMapCountry
4	an EarthMapCountry
5	an EarthMapCountry
6	an EarthMapCountry
7	an EarthMapCountry
8	an EarthMapCountry
9	an EarthMapCountry
10	an EarthMapCountry
11	an EarthMapCountry

```
self
```

Figure 1-5 A dry list of countries.

```
| col |
col := OrderedCollection new.
(XMLDOMParser parse: 'world.svg' asFileReference readStream contents)
  document nodes first nodes
    do: [ :node | (node class = XMLElement)
        ifTrue: [ col add: ( EarthMapCountry new fromXML: node) ]].
col inspect
```

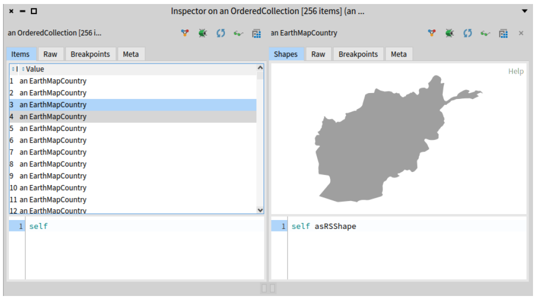
Then you can see a list of countries as shown in Figure 1-5.

1.11 Empowering developers in action

Ok we see the list of objects in the inspector. but we can do better, we want to see **in** the inspector the shape of the country. For this, we define the following method that extends the inspector. You should get the situation shown by Figure 1-6.

```
EarthMapCountry >> inspectorShape: aBuilder
  <inspectorPresentationOrder: 0 title: 'Shape'>

  | canvas |
  canvas := RSCanvas new.
  canvas add: self asRSShape.
  canvas @ RSCanvasController.
  ^ SpRoassalInspectorPresenter new canvas: canvas; yourself
```

Since looking at a list of similar elements does not give any useful information we can simply enhance the experience by adding a little `printOn:` method as follows:



Inspector on an OrderedCollection [256 items] (an ...

an OrderedCollection [256 i...

Items | Raw | Breakpoints | Meta

```
Value
1  an EarthMapCountry
2  an EarthMapCountry
3  an EarthMapCountry
4  an EarthMapCountry
5  an EarthMapCountry
6  an EarthMapCountry
7  an EarthMapCountry
8  an EarthMapCountry
9  an EarthMapCountry
10 an EarthMapCountry
11 an EarthMapCountry
12 an EarthMapCountry
```

```
self
```

an EarthMapCountry

Shapes | Raw | Breakpoints | Meta

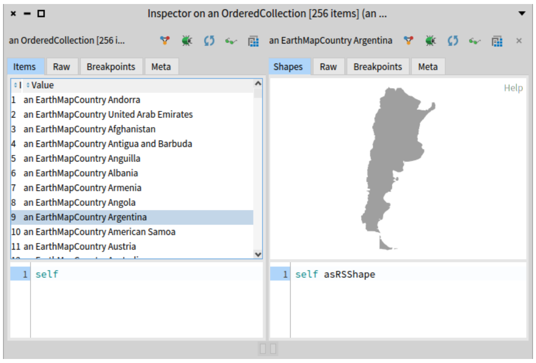
Help

```
self asRSShape
```

Figure 1-6 Extending the inspector.

```
EarthMapCountry >> printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' ', name
```

We can now see and make sense of the list as shown in Figure 1-7.



Inspector on an OrderedCollection [256 items] (an ...

an OrderedCollection [256 i...

Items | Raw | Breakpoints | Meta

```
Value
1  an EarthMapCountry Andorra
2  an EarthMapCountry United Arab Emirates
3  an EarthMapCountry Afghanistan
4  an EarthMapCountry Antigua and Barbuda
5  an EarthMapCountry Anguilla
6  an EarthMapCountry Albania
7  an EarthMapCountry Armenia
8  an EarthMapCountry Angola
9  an EarthMapCountry Argentina
10 an EarthMapCountry American Samoa
11 an EarthMapCountry Austria
```

```
self
```

an EarthMapCountry Argentina

Shapes | Raw | Breakpoints | Meta

Help

```
self asRSShape
```

Figure 1-7 List of counties with printing information and shape displayed.

You can enhance the `printOn:` method to display the country code.

1.12 Introduce the World

We used the following snippet of code but this is a bit brittle.

```
| col |
col := OrderedCollection new.
(XMLDOMParser parse: 'world.svg' asFileReference readStream contents)
  document nodes first nodes
    do: [ :node | (node class = XMLElement)
        ifTrue: [ col add: (EarthMapCountry new fromXML: node) ]].
col
```

We could do better. We could define a class that holds countries. We will define a new class whose responsibilities will be to import the country list and act as a mini database for the future functionalities we want to implement (such as a flag browser).

Defining the map country

```
Object << #EarthMap
  slots: { #countries };
  package: 'EarthTutorial'
```

We initialize the `countries` instance variable to an `OrderedCollection`. Define an accessor.

```
EarthMap >> initialize

  super initialize.
  countries := OrderedCollection new
```

We define the method `importCountryFromXMLNode:` that converts an XML node representing a country into a country object.

```
EarthMap >> importCountryFromXMLNode: aXMLElement

  countries add: (EarthMapCountry new fromXML: aXMLElement)
```

We define the method `xmlTreeFromFile:` that given a file name returns the corresponding XML tree

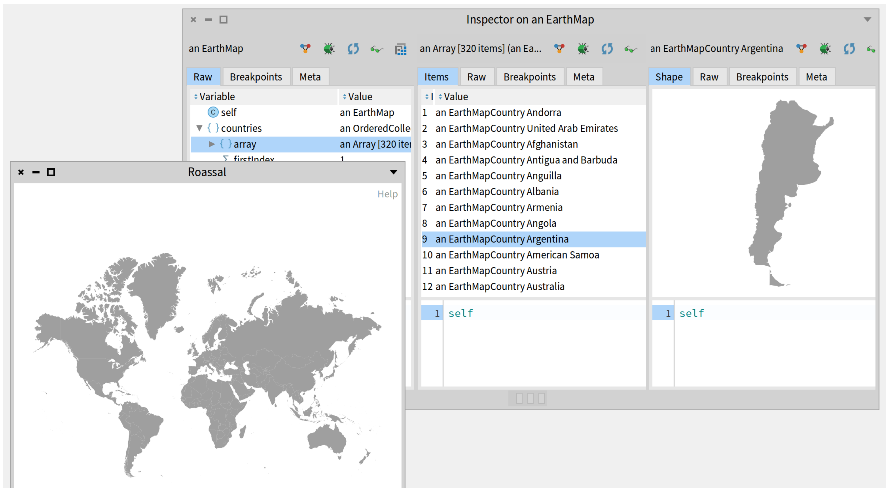
```
EarthMap >> xmlTreeFromFile: aFileName

  ^ aFileName asFileReference readStreamDo: [ :stream |
    (XMLDOMParser parse: stream) document ]
```

```
EarthMap >> populatedCanvas

  ^ RSCanvas new
      addAll: (countries collect: [ :country | country asRSShape ]);
      @ RSCanvasController;
      yourself
```

```
EarthMap >> openPopulatedCanvas

  self populatedCanvas open
```

You have now all the pieces to define the method ct{importCountriesFrom:} that loads an xml file and populate the map with country objects. Once you define it you should get ready to get a map and display it. The following snippet returns a map with filed-up countries and opens a Roassal canvas displaying all the countries.

```
EarthMap new
  importCountriesFrom: (FileSystem workingDirectory / 'pharo-local' /
    'iceberg' / 'EarthTutorial' / 'resources' /'world.svg' );
  openPopulatedCanvas;
  yourself
```



```
Inspector on an EarthMap

an EarthMap
Variable        Value
self            an EarthMap
countries       an OrderedColle
  array         an Array [320 ite
  firstIndex    1

an Array [320 items] (an Ea...
I  Value
1  an EarthMapCountry Andorra
2  an EarthMapCountry United Arab Emirates
3  an EarthMapCountry Afghanistan
4  an EarthMapCountry Antigua and Barbuda
5  an EarthMapCountry Anguilla
6  an EarthMapCountry Albania
7  an EarthMapCountry Armenia
8  an EarthMapCountry Angola
9  an EarthMapCountry Argentina
10 an EarthMapCountry American Samoa
11 an EarthMapCountry Austria
12 an EarthMapCountry Australia

self

an EarthMapCountry Argentina

self
```

Figure 1-8 Getting a Map object and a canvas showing its contents.

1.13 A little note about our process

We defined the previous methods one by one. Notice that we could have started from the original code snippet, turn it into a method, and apply multiple extract methods and other refactorings.

1.14 Grabbing flags from flagcdn

We are ready to work on a country browser displaying its name, flag, and shape. The first thing is to get a flag. For this, we will use the https://flagcdn.com/

The following script gives you the principle to get a flag in PNG from the web. Adapt it to get the flag of the country of your choice. Notice that the `<code>` is expected in lowercase (i.e., fr and not FR).

```
| request pngArray |
request := ZnClient new.
request get: 'https://flagcdn.com/w320/<code>.png'.
request isSuccess ifTrue: [
  pngArray := request response contents ].
pngArray
```

The following expression creates a bitmap image from a byte-array representing a png `ImageReadWriter formFromStream: (ReadStream on: pngArray))`. Use it with the previous snippet to be able to display a flag in PNG format.

1.15 Spec user interface

We will now define a simple user interface to display the list of countries and when we click on one it shows the tag and the flag (See Figure 1-9).

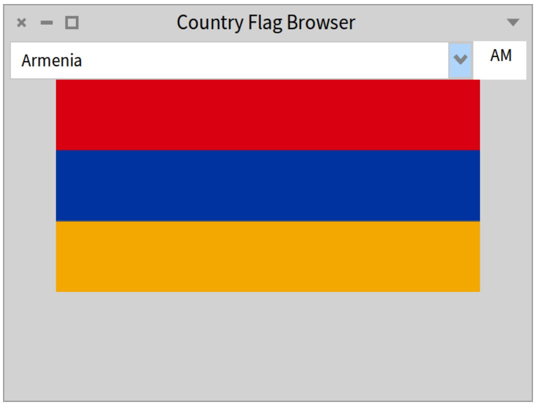


Figure 1-9 The flag browser.

New presenter.

First we define a new class EarthCountryBrowser. It inherits from the class SpPresenterWithModel.

```
SpPresenterWithModel << #EarthCountryBrowser
  slots: { #countryList . #countryCode . #countryFlag };
  package: 'EarthTutorial'
```

The instance variables are:

- countryList is a presenter of the list of country names. It is a drop-list presenter.
- countryCode is a presenter to display the country code such as FR or CH. It is an input field presenter.
- countryFlag is a presenter for the flag. It is an image presenter.

Initialize sub components.

With the method initializePresenters, we initialize the different presenters that compose our interface. Note that the expression self model is given by the superclass SpPresenterWithModel. The model is automatically set using the message ct{on:} as shown below in expression ct{(EarthCountryBrowser on: ...}. In our scenario the model is an instance of the class ct{EarthMap}.

```
EarthCountryBrowser >> initializePresenters

  super initializePresenters.

  countryList := self newDropList.
  countryList display: [ :item | item name ].
  countryList sortingBlock: [ :a :b | a model name < b model name ].
  countryList items: self model countries.

  countryCode := self newTextInput.
  countryCode editable: false.
  countryCode text: '   --   '.

  countryFlag := self newImage
```

Layout the elements.

We set the placement of the subcomponents by defining the method defaultLayout.

```
EarthCountryBrowser >> defaultLayout

  ^ SpBoxLayout newTopToBottom
      add: (SpBoxLayout newLeftToRight
           add: countryList expand: true;
           add: countryCode width: 40)
      height: self class toolbarHeight;
      add: countryFlag height: 350;
      yourself
```

At this stage you should be able to open the browser even if it is not fully working.

Here is the snippet we used during this tutorial.

```
(EarthCountryBrowser on:
  (EarthMap new importCountriesFrom: (FileSystem workingDirectory /
    'pharo-local' / 'iceberg' / 'EarthTutorial' / 'resources'
    /'world.svg' )))
  open
```

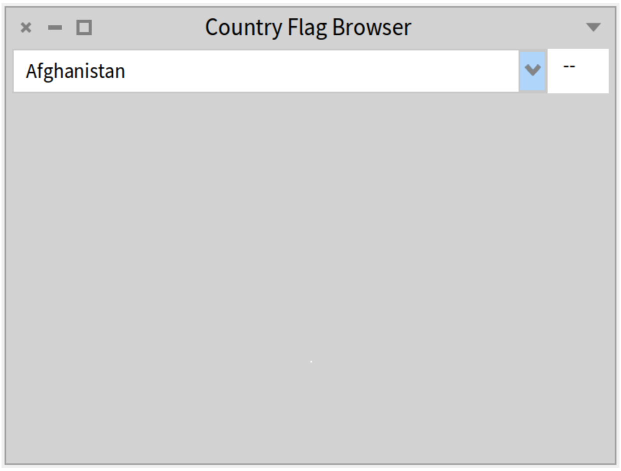


Figure 1-10 A first version of the interface.

1.16 Enhancing the browser

We turn the snippet that we used to fetch flags into a method named `flagForCountryCode:`. Notice that if the request fails we return a blue red rectangle.

```
EarthCountryBrowser >> flagForCountryCode: astring

  | request pngArray |
  request := ZnClient new.
  request get:
    'https://flagcdn.com/w320/' , astring asLowercase , '.png'.
  request isSuccess ifTrue: [
    pngArray := request response contents.
    ^ ImageReadWriter formFromStream: (ReadStream on: pngArray) ].
  ^ BorderedMorph new asForm
```

Once this is done, we can now define the method `onCountrySelected:` that will display the country code and the flag. We concatenate some spaces in front of the country code so that it looks better on the screen.

```
EarthCountryBrowser >> onCountrySelected: countryItem

  countryCode text: '     ' , countryItem code.
  self showFlag: countryItem code
```

The method `onCountrySelected:` will be invoked each time the user selects a new country. The method `connectPresenters` is responsible for defining the interaction between the elements. Here we simply invoke the method `onCountrySelected:` each time a new country is selected.

```
EarthCountryBrowser >> connectPresenters

  countryList whenSelectedItemChangedDo: [ :item |
        self onCountrySelected: item ].
```

We let as an exercise the display of the roassal visualization. For this you can add a new component to the browser and initialize it to `newRoassal`. The message `newRoassal` creates an instance of the class `SpRoassalPresenter`. You specify set and get the canvas (using the messages `canvas` and `canvas:`) to be displayed.

1.17 Conclusion

In this little tutorial, we show several important aspects of Pharo.

- First, in a couple of lines we created a little tool. We took the time to decompose scripts into objects and gave such objects responsibilities (such as import, conversion, ...)
- Second, we show how we can dabble complex objects and navigate easily in their structure.

- Third, we show that we can easily extend the tools of the environment to get more information about our own data.

All this makes Pharo a really productive platform to model business while improving the feedback loops and speed to manipulate data.