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

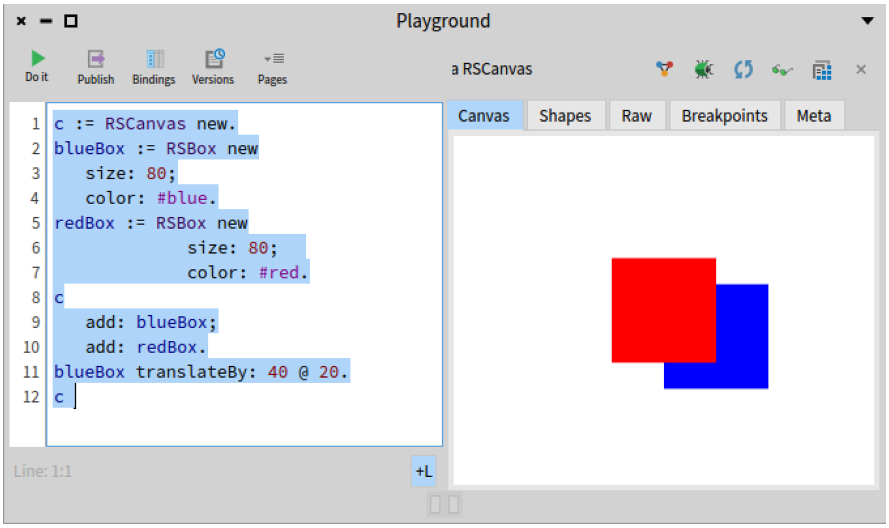
This tutorial uses the following resources:

- 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.

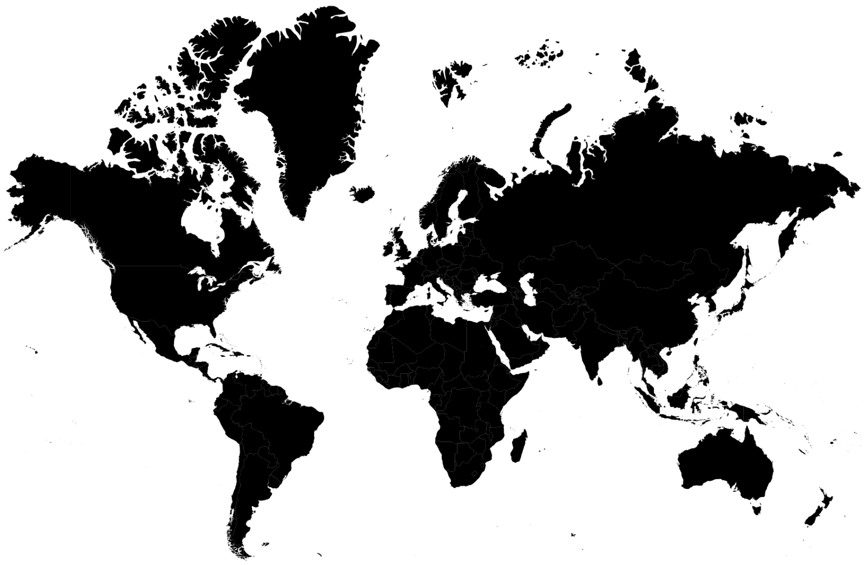


**Figure 1-2** World.svg rendered.

## 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.

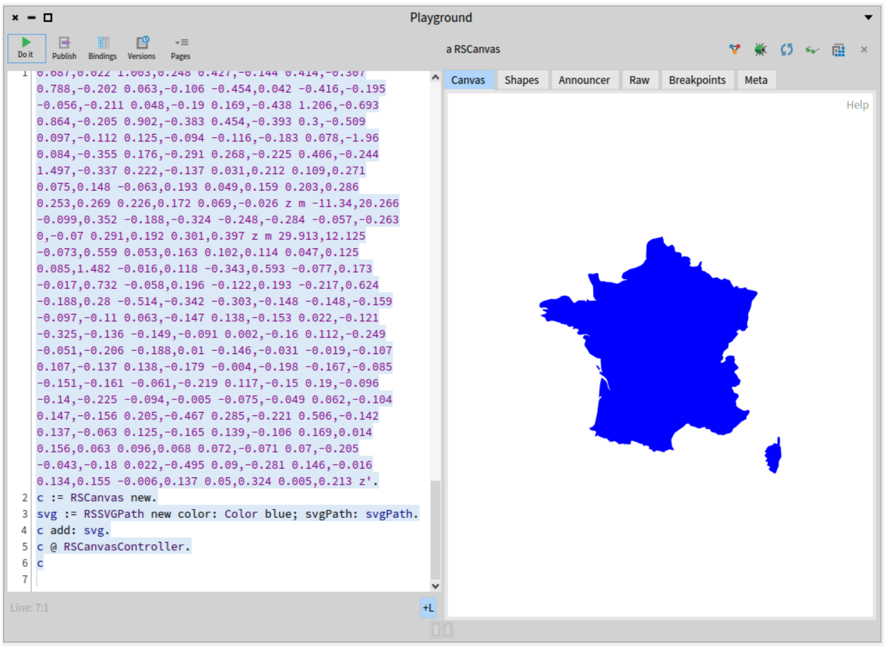
```
| c svg |
c := RSCanvas new.
svg := (RSSVGPath new
  svgPath: 'M 100 350 q 150 -300 300 0';
  yourself).
c addShape: svg.
c @ RSCanvasController.
c
```

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`.

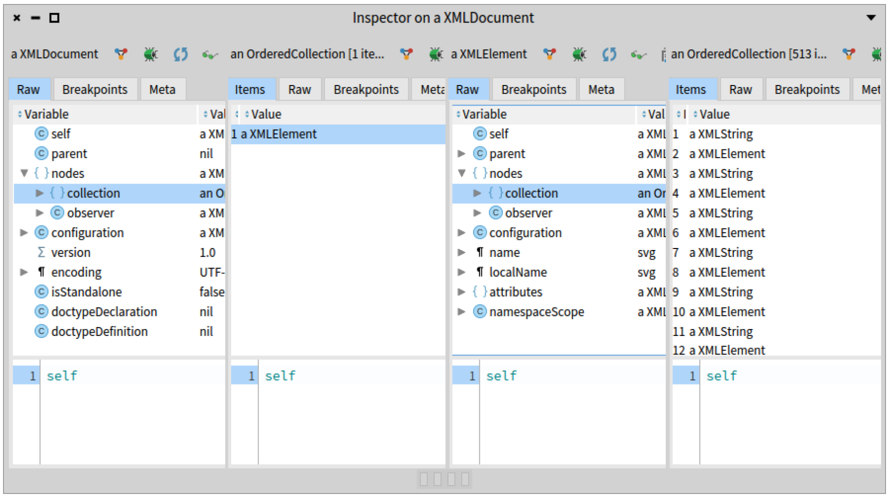
```
Metacello new
  baseline: 'XMLParser';
  repository: 'github://pharo-contributions/XML-XMLParser/src';
  load.
```

## 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.



Inspector on a XMLDocument

a XMLDocument | an OrderedCollection [1 ite... | a XMLElement | an OrderedCollection [513 i...

| 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 |

| | Value |
|---|---|
| 1 | a XMLElement |

| 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 |

| 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
```

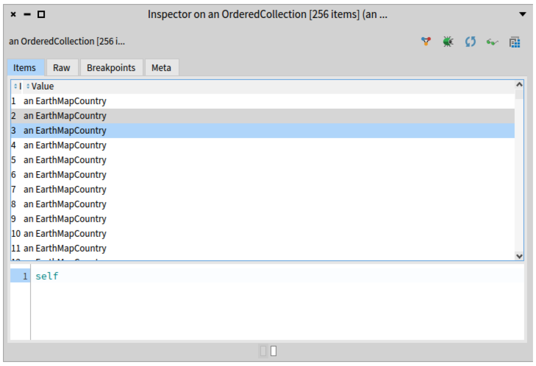
## 1.10 Grabbing a first country

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
```

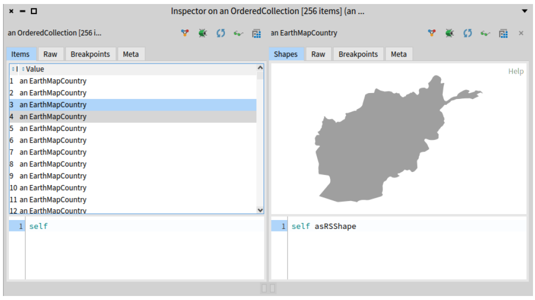
Then you can see a list of countries as shown in Figure 1-5.

## 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
  <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...
an EarthMapCountry
Items
Raw
Breakpoints
Meta
Shapes
Help
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
```

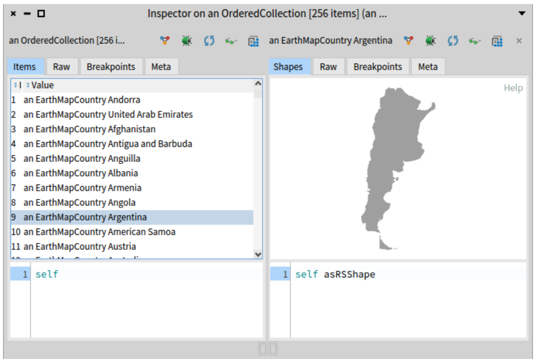
```
self
```

```
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...
an EarthMapCountry Argentina
Items
Raw
Breakpoints
Meta
Shapes
Help
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
```

```
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`.

```
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

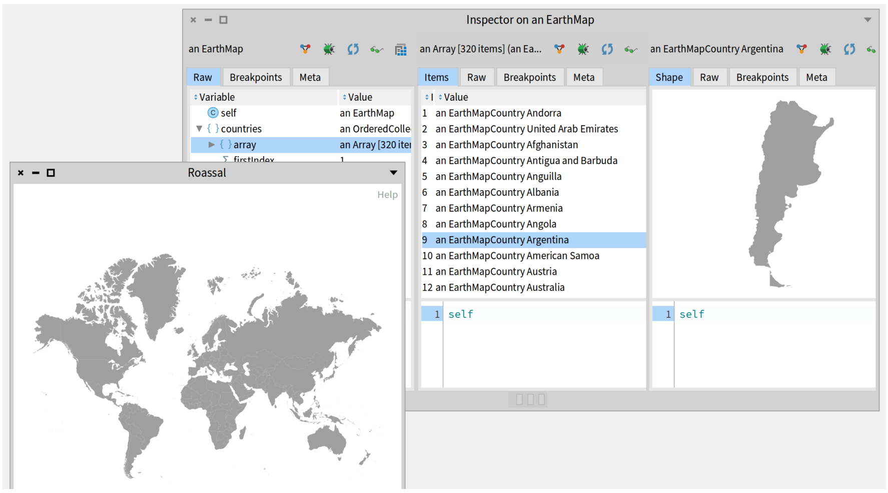
```
EarthMap >> xmlTreeFromFile: aFileName

  ^ 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
```

Now we are 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
```



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

| 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
```

```
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).

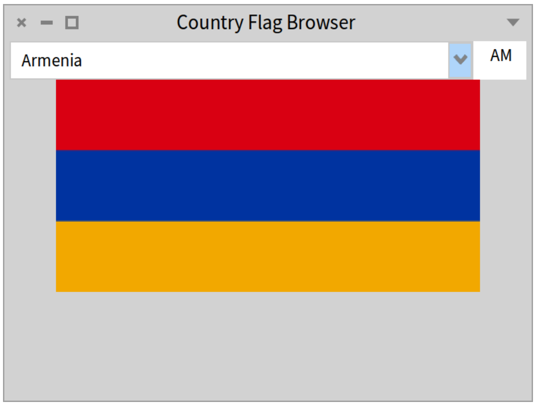


**Figure 1-9** The flag browser.

### 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.
- map is an instance of the class `EarthMap`.

### Initialize sub components.

With the method `initializePresenters`, we initialize the different presenters that compose our interface.

```
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
```

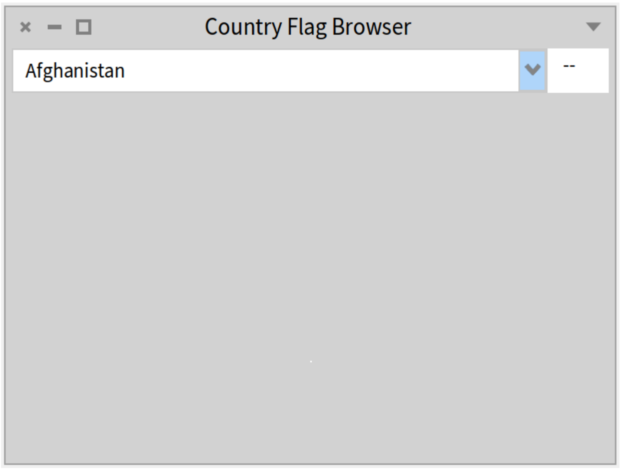


**Figure 1-10** A first version of the interface.

## 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 done we can know 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.

CHAPTER 2

# A simple network simulator with payload

In this chapter, we will develop a simulator for a computer network, from scratch, step by step. The program starts with a simplistic model of a computer network, made of objects that represent different parts of a local network such as packets, nodes, links, workstations, and hubs.

At first, we will just simulate the different steps of packet delivery and have fun with the system. As a second step, we will extend the basic functionalities by adding extensions such as a hub and different packet routing strategies. In doing so we will revisit many object-oriented concepts such as polymorphism, encapsulation, and hooks and templates. Finally, this system could be further refined to become an experimentation platform to explore and understand networking and distributed algorithms.

> **Note** that this version of the exercise is more abstract than the one in the new version of the book: *Learning OOD with TDD in Pharo* and *Advanced Micro Projects* (the book available with the Advanced Design Mooc).

## Basic definitions and a starting point

We need to establish a basic mental model, so what does the description above tell us about a network? A *network* is a number of interconnected *nodes*, which exchange *data packets*. We will therefore probably need to model the nodes, the connection links, and the packets. Let's fill out a little more detail:

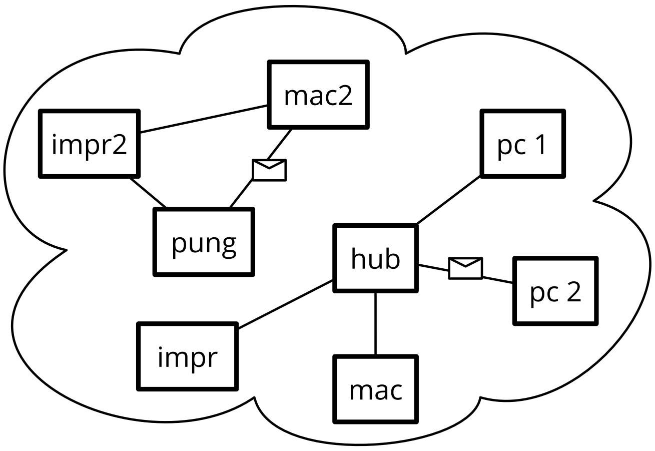


**Figure 2-1** Two little networks composed of nodes and sending packets over links.

- Nodes have *addresses*, and can send and receive packets.
- *Links* connect two nodes together, and transmit the packets between them.
- A *packet* transports a payload and includes the address of the node to which it should be delivered; if we want nodes to be able to answer (after they receive the packet), packets should also have the address of the node that originally sent it.

## 2.1 Packets are simple value objects

Packets seem to be the simplest objects in our model: we only need to create them, ask them about the data they contain, and... that's about it. Once created, a packet object is a passive data structure; it will not change its data, knows nothing of the surrounding network, and has no real behavior to speak of. This sort of object is often referred to as a *value object*.

Let's start by defining a test class and a first test, sketching out what creating and looking at packets should look like:

```
TestCase << #KANetworkEntitiesTest
    package: 'NetworkSimulator-Tests'
```

```
KANetworkEntitiesTest >> testPacketCreation
    | src dest payload packet |
    src := Object new.
    dest := Object new.
    payload := Object new.

    packet := KANetworkPacket from: src to: dest payload: payload.

    self assert: packet sourceAddress equals: src.
    self assert: packet destinationAddress equals: dest.
    self assert: packet payload equals: payload
```

By writing this unit test, we have described how we think packets should be created, using a `from:to:payload:` constructor message, and how they should be accessed, using three messages: `sourceAddress`, `destinationAddress`, and `payload`. Since we have not yet decided what addresses and payloads should look like, we will just pass arbitrary objects as parameters. All that matters is that when we tell the packet to give us one of those three pieces of information, it returns the correct object back.

Of course, if we compile and run this test method, it will fail because the class `KANetworkPacket` has not been created yet, nor have any of the four above messages. You can either execute the test and let the system prompt you to define the missing objects and messages when needed or we can define the class from the start:

```
Object << #KANetworkPacket
    slots: { #sourceAddress . #destinationAddress . #payload};
    package: 'NetworkSimulator-Core'
```

The class-side constructor method creates an instance which it returns after sending it an initialization message, so nothing too original as far as constructors go:

```
KANetworkPacket class >> from: sourceAddress to: destinationAddress
    payload: anObject
    ... Your code ...
```

That constructor will need to pass the initialization parameters to the new instance. It's preferable to define a single initialization method that takes all needed parameters at once, since it is only supposed to be called when creating packets and should not be confused with a setter:

```
KANetworkPacket >> initializeSource: *sourceAddress destination:
    destinationAddress payload: aPayload
    ... Your code ...
```

Once a packet is created, all we need to do with it is to obtain its payload, or the addresses of its source or destination nodes. Define the following getters:

```
KANetworkPacket >> sourceAddress
  ... Your code ...
KANetworkPacket >> destinationAddress
  ... Your code ...
KANetworkPacket >> payload
  ... Your code ...
```

Now our test should be running and passing. That's enough for our admittedly simplistic model of packets. We've completely ignored the layers of the OSI model, but it could be an interesting exercise to model them more precisely.

## 2.2 Nodes are known by their address

The first obvious thing we can say about a network node is that if we want to be able to send packets to it, then it should have an address. Let's translate that into a test:

```
KANetworkEntitiesTest >> testNodeCreation
    | address node |
    address := Object new.
    node := KANetworkNode withAddress: address.
    self assert: node address equals: address
```

Like before, to run this test to completion, we will have to define the `KANetworkNode` class:

```
Object << #KANetworkNode
    slots: { #address};
    package: 'NetworkSimulator-Core'
```

Then a class-side constructor method takes the address of the new node as its parameter:

```
KANetworkNode class >> withAddress: aNetworkAddress
    ^ self new
        initializeAddress: aNetworkAddress
```

The constructor relies on an instance-side initialization method, and the test asserts that the `address` accessor works as expected. Let's define them:

```
KANetworkNode >> initializeAddress: aNetworkAddress
  ... Your code ...
```

```
KANetworkNode >> address
  ... Your code ...
```

Again, our tests should now pass.

## 2.3 Links are one-way connections between nodes

After nodes and packets, what about looking at links? In the real world, network cables are bidirectional, but that's because they have wires going both ways. Here, we're going to keep it simple and define links as simple one-way connections; to make a two-way connection, we will just use two links, one in each direction.

However, creating links that know their source and destination node is not sufficient: nodes also need to know about their outgoing links, otherwise they cannot send packets. Let's write a test to cover this.

```
KANetworkEntitiesTest >> testNodeLinking
    | node1 node2 link |
    node1 := KANetworkNode withAddress: #address1.
    node2 := KANetworkNode withAddress: #address2.
    link := KANetworkLink from: node1 to: node2.

    link attach.

    self assert: (node1 hasLinkTo: node2)
```

This test creates two nodes and a link. After telling the link to *attach* itself, we check that it did so: the source node should confirm that it has an outgoing link to the destination node. Note that the constructor could have registered the link with `node1`, but we opted for a separate message `attach` instead, because it's bad practice to have a constructor change other objects. This way we can build links between arbitrary nodes and still have control of when the connection really becomes part of the network model. For symmetry, we could have specified that `node2` has an incoming link from `node1`, but that ends up not being necessary, so we leave that out for now.

Again, we need to define the class of links:

```
Object << #KANetworkLink
    slots: { #source . #destination};
    package: 'NetworkSimulator-Core'
```

A constructor that passes the two required parameters to an instance-side initialization message:

```
KANetworkLink class >> from: sourceNode to: destinationNode
    ^ self new
        initializeFrom: sourceNode to: destinationNode
```

As well as the initialization method and accessors:

```
KANetworkLink >> initializeFrom: sourceNode to: destinationNode
  ... Your code ...
KANetworkLink >> source
  ... Your code ...
KANetworkLink >> destination
  ... Your code ...
```

The `attach` method of a link should not (and cannot) directly modify the source node, so it must delegate to it instead.

```
KANetworkLink >> attach
    source attach: self
```

This is an example of separation of concerns: the link knows which node has to do what, but only the node itself knows precisely how to do that. If a node knows about all its outgoing links then it means it has a collection of them, and attaching a link adds it to that collection:

```
KANetworkNode >> attach: anOutgoingLink
    outgoingLinks add: anOutgoingLink
```

For this method to compile correctly, we will need to extend `KANetworkNode` with the new instance variable `outgoingLinks`, and with the corresponding initialization code:

```
KANetworkNode >> initialize
    outgoingLinks := Set new.
```

And finally the unit test relied on a predicate method which we define in `KANet-workNode`:

```
KANetworkNode >> hasLinkTo: anotherNode
    ... Your code ...
```

The method `hasLinkTo:` should verify that there is at least one outgoing link whose destination is the node passed in as an argument. We suggest to have a look at the iterator `anySatisfy:` to express this logic.

Again, all the tests should now pass.

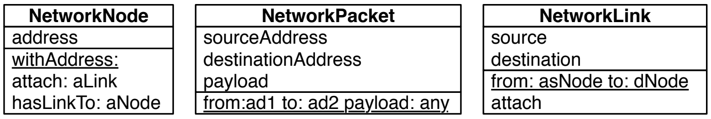


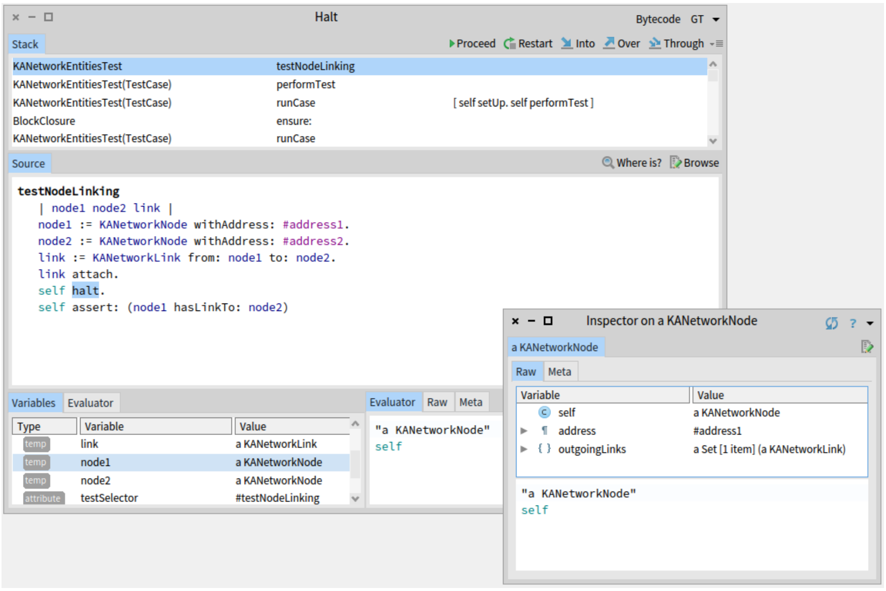
**Figure 2-2** Current API of our three main classes.

## 2.4 Making our objects more understandable

When programming we often make mistakes and it is important to help developers to address them. Let us put a breakpoint in to try and understand the objects we have been working with:

```
KANetworkEntitiesTest >> testNodeLinking
  | node1 node2 link |
  node1 := KANetworkNode withAddress: #address1.
  node2 := KANetworkNode withAddress: #address2.
  link := KANetworkLink from: node1 to: node2.
  link attach.
  self halt.
  self assert: (node1 hasLinkTo: node2)
```

Running the test will open a debugger as the one shown in Figure 2-3. We can see objects, but their textual representation is too generic to be of any help.



| | | |
|---|---|---|
| KANetworkEntitiesTest | testNodeLinking | |
| KANetworkEntitiesTest(TestCase) | performTest | |
| KANetworkEntitiesTest(TestCase) | runCase | [ self setUp. self performTest ] |
| BlockClosure | ensure: | |
| KANetworkEntitiesTest(TestCase) | runCase | |

```
testNodeLinking
   | node1 node2 link |
   node1 := KANetworkNode withAddress: #address1.
   node2 := KANetworkNode withAddress: #address2.
   link := KANetworkLink from: node1 to: node2.
   link attach.
   self halt.
   self assert: (node1 hasLinkTo: node2)
```

| Type | Variable | Value |
|---|---|---|
| temp | link | a KANetworkLink |
| temp | node1 | a KANetworkNode |
| temp | node2 | a KANetworkNode |
| attribute | testSelector | #testNodeLinking |

```
"a KANetworkNode"
self
```

| Variable | Value |
|---|---|
| self | a KANetworkNode |
| address | #address1 |
| outgoingLinks | a Set [1 item] (a KANetworkLink) |

```
"a KANetworkNode"
self
```

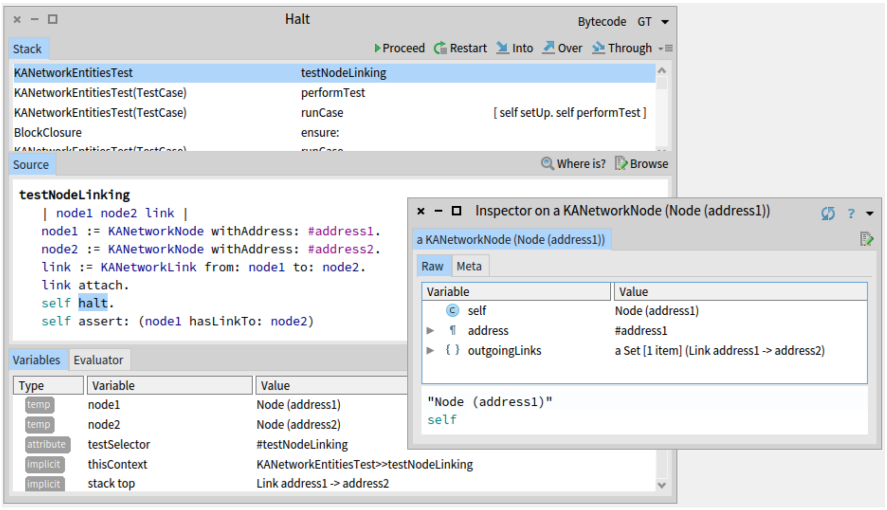
**Figure 2-3** Navigating specific objects having a generic presentation.

The method `printOn:` is responsible for printing the object's representation. So we should redefine this method for our objects:

```
KANetworkNode >> printOn: aStream
  aStream nextPutAll: 'Node ('.
  aStream nextPutAll: address , ')'
```

```
KANetworkLink >> printOn: aStream
  aStream nextPutAll: 'Link'.
  source
    ifNotNil: [ aStream
        nextPutAll: ' ';
        nextPutAll: source address ].
  destination
    ifNotNil: [ aStream
        nextPutAll: ' -> ';
        nextPutAll: destination address ]
```

Now if we rerun the test we obtain a better user experience as shown in Figure 2-4: we can see the address of a node and the source and destination of a link.



```
KANetworkEntitiesTest              testNodeLinking
KANetworkEntitiesTest(TestCase)    performTest
KANetworkEntitiesTest(TestCase)    runCase        [ self setUp. self performTest ]
BlockClosure                       ensure:
```

```
testNodeLinking
    | node1 node2 link |
    node1 := KANetworkNode withAddress: #address1.
    node2 := KANetworkNode withAddress: #address2.
    link := KANetworkLink from: node1 to: node2.
    link attach.
    self halt.
    self assert: (node1 hasLinkTo: node2)
```

| Type | Variable | Value |
|---|---|---|
| temp | node1 | Node (address1) |
| temp | node2 | Node (address2) |
| attribute | testSelector | #testNodeLinking |
| implicit | thisContext | KANetworkEntitiesTest>>testNodeLinking |
| implicit | stack top | Link address1 -> address2 |

| Variable | Value |
|---|---|
| self | Node (address1) |
| address | #address1 |
| outgoingLinks | a Set [1 item] (Link address1 -> address2) |

```
"Node (address1)"
self
```

**Figure 2-4** Navigating objects offering a customized presentation.

## 2.5 Simulating the steps of packet delivery

The next big feature is that nodes should be able to send and receive packets, and links should be able to transmit them.

```
KANetworkEntitiesTest >> testSendAndTransmit
    | srcNode destNode link packet |
    srcNode := KANetworkNode withAddress: #src.
    destNode := KANetworkNode withAddress: #dest.
    link := (KANetworkLink from: srcNode to: destNode) attach;
    yourself.
    packet := KANetworkPacket from: #address to: #dest payload:
    #payload.

    srcNode send: packet via: link.
    self assert: (link isTransmitting: packet).
    self deny: (destNode hasReceived: packet).

    link transmit: packet.
    self deny: (link isTransmitting: packet).
    self assert: (destNode hasReceived: packet)
```

We create and setup two nodes, a link between them, and a packet. Now, to control which packets get delivered in which order, we specify that it happens in separate, controlled steps. This will allow us to model packet delivery precisely, to simulate latency, out-of-order reception, etc.

- First, we tell the node to send the packet using the message `send:via:`. At that point, the packet should be passed to the link for transmission, but not be delivered yet.
- Then, we tell the link to actually transmit the packet along using the message `transmit:`, and thus the packet should be received by the destination node.

## 2.6 Sending a packet

To send a packet, the node emits it on the link:

```
KANetworkNode >> send: aPacket via: aLink
    aLink emit: aPacket
```

For the simulation to be realistic, we do not want the packet to be delivered right away. Instead, emitting a packet just stores it in the link until the user selects this packet be transmitted using the `transmit:` message. Storing packets requires adding an instance variable to `KANetworkLink`, as well as specifying how this instance variable should be initialized.

```
Object << #KANetworkLink
    slots: {#source . #destination . #packetsToTransmit};
    package: 'NetworkSimulator-Core'
```

```
KANetworkLink >> initialize
    packetsToTransmit := OrderedCollection new
```

```
KANetworkLink >> emit: aPacket
    "Packets are not transmitted right away, but stored.
    Transmission is explicitly triggered later by sending #transmit:."

    packetsToTransmit add: aPacket
```

We also add a testing method to check whether a given packet is currently being transmitted by a link:

```
KANetworkLink >> isTransmitting: aPacket
    ... Your code ...
```

## 2.7 Transmitting across a link

Transmitting a packet means telling the link's destination node to receive it. Nodes only consume packets addressed to them; fortunately this is what will happen in our test, so we can worry about the alternative case later (`notYetImplemented` is a special message that we can use in place of code that we will have to write eventually, but prefer to ignore for now).

```
KANetworkNode >> receive: aPacket from: aLink
    aPacket destinationAddress = address
        ifTrue: [
            self consume: aPacket.
            arrivedPackets add: aPacket ]
        ifFalse: [ self notYetImplemented ]
```

Consuming a packet represents what the node will do with it at the application level; for now let's just define an empty `consume:` method, as a placeholder:

```
KANetworkNode >> consume: aPacket
    "Default handling is to do nothing."
```

After consuming the packet, we note that it did arrive; this is mostly for testing and debugging, but someday we might want to simulate packet losses and reemissions. Don't forget to declare and initialize the `arrivedPackets` instance variable, along with its accessor:

```
KANetworkNode >> hasReceived: aPacket
  ... Your code ...
```

Now we can implement the `transmit:` message. A link can not transmit packets that have not been sent via it, and once transmitted, the packet should not be in the link anymore. We should remove it from the link's list of packets to be transmitted and tell the destination to receive it using the message `receive:from:`.

```
KANetworkLink >> transmit: aPacket
    "Transmit aPacket to the destination node of the receiver link."
    ... Your code ...
```

At this point all our tests should pass.

Note that the message `notYetImplemented` is not called, since our tests do not yet require routing. Figure 2-5 shows that the API of our classes is getting richer than before.

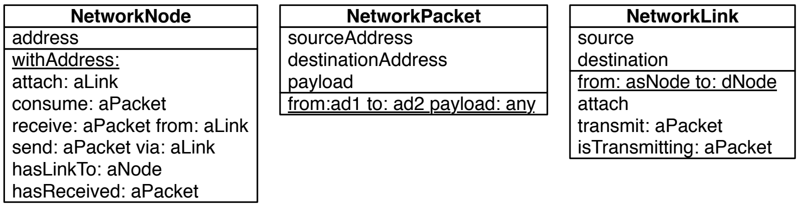


**Figure 2-5** Richer API.

## 2.8 The loopback link

On a real network, when a node wants to send a packet to itself, it does not need any connection to do so. In real-world networking stacks, loopback routing shortcuts the lower networking layers, but this is more detail than we are modeling here.

Still, we do want to model the fact that the loopback link is a little special, so each node will store its own loopback link, separately from the outgoing links. We start to define a test.

```
KANetworkEntitiesTest >> testLoopback
    | node packet |
    node := KANetworkNode withAddress: #address.
    packet := KANetworkPacket from: #address to: #address payload:
    #payload.

    node send: packet.
    node loopback transmit: packet.

    self assert: (node hasReceived: packet).
    self deny: (node loopback isTransmitting: packet)
```

The loopback link is implicitely created as part of the node itself. We also introduce a new `send:` message, which takes the responsibility of selecting the link to emit the packet. For triggering packet transmission, we have to use a specific accessor get to the loopback link of the node.

First, we have to add yet another instance variable in nodes:

```
Object << #KANetworkNode
    slots: {#address . #outgoingLinks .  #loopback . #arrivedPackets};
    package: 'NetworkSimulator-Core'
```

As with all instance variables, we have to remember to make sure it is correctly initialized; we thus modify `initialize`:

```
KANetworkNode >> initialize
    ... Your code ...
```

The accessor has nothing special:

```
KANetworkNode >> loopback
    ^ loopback
```

And finally we can focus on the `send:` method and automatic link selection. The method `send:` should be more generic than the method `send:via:` and will be one exposed as a public entry point.

This method has to rely on some routing algorithm to identify which links will transmit the packet closer to its destination. Since some routing algorithms select more than one link, we will implement routing as an *iteration* method, which evaluates the given block for each selected link.

```
KANetworkNode >> send: aPacket
    "Send aPacket, leaving the responsibility for routing to the node."
    self
        linksTowards: aPacket destinationAddress
        do: [ :link | self send: aPacket via: link ]
```

One of the simplest routing algorithms is *flooding*, which just sends the packet to every single outgoing link. Obviously this is a waste of bandwidth, but it works without any knowledge of the network topology beyond the list of outgoing links.

There is, however, one case where we know how to route the packet: if the destination address matches the one of the current node, we can select the loopback link. The logic of `linksTowards:do:` should look something like:

- compare the packet's destination address with the one of the node
- if it is the same, we execute the block using the loopback link
- else we simply iterate on the outgoing links of the receiver.

```
KANetworkNode >> linksTowards: anAddress do: aBlock
    "Simple flood algorithm: route via all outgoing links.
    However, just loopback if the receiver node is the routing
     destination."
    ... Your code ...
```

Now we have the basic model working we can try more realistic examples.

## 2.9 Modeling the network itself

More realistic tests will require non-trivial networks. We therefore need an object that represents the network as a whole, to avoid keeping many nodes and links in individual variables. We will introduce a new class `KANetwork`, whose responsibility is to help us build, assemble then find the different nodes and links involved in a network.

Let’s start by creating another test class, to keep things in order:

```
TestCase << #KANetworkTest
    slots: { #net . #hub . #alone};
    package: 'NetworkSimulator-Tests'
```

Since every test will need to rebuild the example network from scratch, we’ll move those steps out to a method called `buildNetwork`. We’ll then send that message in the `setUp` method of our test, which is called before each test method is run.

```
KANetworkTest >> setUp
    super setUp.
    self buildNetwork
```

We’ll worry about the implementation of `buildNetwork` shortly.

Before we go any further with the test setup, let’s write a test that will pass once we’ve made progress; we want to be able to access network nodes given their addresses. Here we check that we get a the node `hub` based on its address:

```
KANetworkTest >> testNetworkFindsNodesByAddress
    self
        assert: (net nodeAt: hub address ifNone: [ self fail ])
        equals: hub
```

We will have to implement `nodeAt:ifNone:` on our `KANetwork` class, but first we need to decide how we’re going to build our example network. This is what it’s going to look like:

- The nodes `pc1`, `pc2`, `mac`, and `impr` connected in a star-shape around a central `hub` node
- A pair of connected nodes, `ping` and `pong`, which are part of the network but not linked to any other nodes
- And the node `alone` which sits all by itself, not even a part of our network

We can picture this as in Figure 2-6.

Expanding a network implies adding new connections and possibly new nodes to it. If the `net` object understands a `connect: aNode to: anotherNode` mes-

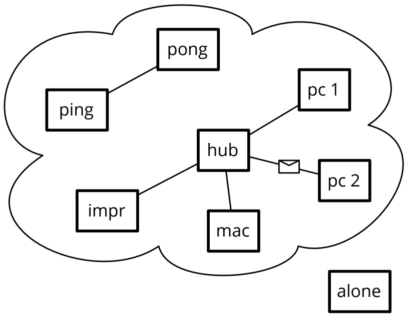


**Figure 2-6** Our network `net`.

sage, we should be able to build nodes and connect them into a network that matches the figure. This can then be the content of our `buildNetwork` method:

```
KANetworkTest >> buildNetwork
  alone := KANetworkNode withAddress: #alone.
  net := KANetwork new.
  hub := KANetworkNode withAddress: #hub.
  #( mac pc1 pc2 impr ) do: [ :addr |
    | node |
    node := KANetworkNode withAddress: addr.
    net connect: node to: hub ].
  net
    connect: (KANetworkNode withAddress: #ping)
    to: (KANetworkNode withAddress: #pong)
```

The name of the `connect:to:` message suggests that establishing the bidirectional links is the responsibility of the `net` object. It also has to remember enough information to allow us to inspect the network topology; we can simply store nodes and links in a couple of sets, even though that representation is a little redundant. Let's define the class with two instance variables:

```
Object << #KANetwork
    slots: {#nodes . #links};
    package: 'NetworkSimulator-Core'
```

Whenever we define an instance variable, initialization (if any) comes next:

```
KANetwork >> initialize
    ... Your code ...
```

Now we should give the network the ability to create links. We'll use this method to add links to the network.

```
KANetwork >> makeLinkFrom: aNode to: anotherNode
  ^ KANetworkLink from: aNode to: anotherNode
```

We will also add a low level method `add:` to add a node in a network:

```
KANetwork >> add: aNode
  nodes add: aNode
```

And to test the network construction we add a little test method:

```
KANetwork >> doesRecordNode: aNode
  ^ nodes includes: aNode
```

Now we can add isolated nodes to the network.

### Connecting nodes.

Connecting nodes without ensuring that they are part of the network doesn't make sense. Therefore, when connecting nodes, we will also:

- ensure that nodes are added to the network (by adding them in the node Set of the network)
- create and attach links to the nodes in *both* directions
- and finally store both links in the network

Here is a test covering connecting nodes in a network:

```
KANetworkTest >> testConnect
  | netw hubb mac pc1 |
  netw := KANetwork new.
  hubb := KANetworkNode withAddress: #hub.
  mac := KANetworkNode withAddress: #mac.
  pc1 := KANetworkNode withAddress: #pc1.

  netw connect: hubb to: mac.
  self assert: (hubb hasLinkTo: mac).
  self assert: (mac hasLinkTo: hubb).
  self assert: (netw doesRecordNode: hubb).
  self assert: (netw doesRecordNode: mac).

  netw connect: hubb to: pc1.
  self assert: (hubb hasLinkTo: pc1).
  self assert: (mac hasLinkTo: hubb)
```

Now implement the `connect:to:` method. For concision, note that the `attach` method we defined previously effectively returns the link.

```
KANetwork >> connect: aNode to: anotherNode
    ... Your code ...
```

The test `testConnect` should now be green.

## 2.10 Looking up nodes

At this point, the test `testNetworkFindsNodesByAddress` should run through `setUp` method, but fail in the unit test itself. This is because we still need to implement node lookup. The base lookup should find the first node that has the requested address, or evaluate a fall-back block (a perfect case for the `detect:ifNone:` message):

```
KANetwork >> nodeAt: anAddress ifNone: noneBlock
    ... Your code ...
```

We can also make a convenience method, `nodeAt:`, for node lookup. This will raise the predefined `NotFound` exception if it does not find the node. Let's first write a test which validates this behavior:

```
KANetworkTest >> testNetworkOnlyFindsAddedNodes
    self
        should: [ net nodeAt: alone address ]
        raise: NotFound
```

Then we can simply express `nodeAt:` by delegating to `nodeAt:ifNone:`. Note that to raise an exception, you can simply send the message `signal` to the exception class. Here we use the specific class method `signalFor:in:` defined on the `NotFound` class. `NotFound` exceptions are used when a collection does not contain an object; the `signalFor:in:` method adds both the collection and the missing object to the exception, generating a more detailed description of the problem for the user:

```
KANetwork >> nodeAt: anAddress
    ^ self
        nodeAt: anAddress
        ifNone: [ NotFound signalFor: anAddress in: self ]
```

## 2.11 Looking up links

Next, we want to be able to lookup links between two nodes. Again we define a new test:

```
KANetworkTest >> testNetworkFindsLinks
    | link |
    self
        shouldnt: [ link := net linkFrom: #pong to: #ping ]
        raise: NotFound.
    self
        assert: link source
        equals: (net nodeAt: #pong).
    self
        assert: link destination
        equals: (net nodeAt: #ping)
```

And we define the method `linkFrom:to:`, returning the link between source and destination nodes with matching addresses, and signalling `NotFound` if no such link is found:

```
KANetwork >> linkFrom: sourceAddress to: destinationAddress
    ... Your code ...
```

## Final check.

As a final check, let's try some of the previous tests, first on the isolated `alone` node, showing that loopback works even without a network connection:

```
KANetworkTest >> testSelfSend
    | packet |
    packet := KANetworkPacket
        from: alone address
        to: alone address
        payload: #something.
    self assert: (packet isAddressedTo: alone).
    self assert: (packet isOriginatingFrom: alone).

    alone send: packet.
    self deny: (alone hasReceived: packet).
    self assert: (alone loopback isTransmitting: packet).

    alone loopback transmit: packet.
    self deny: (alone loopback isTransmitting: packet).
    self assert: (alone hasReceived: packet)
```

You can see that we used new convenience testing methods `isAddressedTo:` and `isOriginatingFrom:` which help inspect the state of a simulated network without explicitly comparing addresses. However, those methods should not take part in network simulation code, since in the real world nodes can never know their peers other than through their addresses.

```
KANetworkPacket >> isAddressedTo: aNode
    ^ destinationAddress = aNode address
```

```
KANetworkPacket >> isOriginatingFrom: aNode
    ^ sourceAddress = aNode address
```

The second test attempts transmitting a packet in the network, between the directly connected nodes `ping` and `pong`:

```
KANetworkTest >> testDirectSend
    | packet ping pong link |
    packet := KANetworkPacket from: #ping to: #pong payload: #ball.
    ping := net nodeAt: #ping.
    pong := net nodeAt: #pong.
    link := net linkFrom: #ping to: #pong.

    ping send: packet.
    self assert: (link isTransmitting: packet).
    self deny: (pong hasReceived: packet).

    link transmit: packet.
    self deny: (link isTransmitting: packet).
    self assert: (pong hasReceived: packet)
```

Both tests should pass with no additional work, since they just reproduce what we already tested in `KANetworkEntitiesTest` and adding `KANetwork` did not impact the established behavior of nodes, links, and packets.

## 2.12 Packet delivery with forwarding

Up until now, we have only tested packet delivery between directly connected nodes; now let's try sending to a node so that the packet has to be forwarded through the `hub`.

```
KANetworkTest >> testSendViaHub
    | hello mac pc1 firstLink secondLink |
    hello := KANetworkPacket from: #mac to: #pc1 payload: 'Hello!'.
    mac := net nodeAt: #mac.
    pc1 := net nodeAt: #pc1.
    firstLink := net linkFrom: #mac to: #hub.
    secondLink := net linkFrom: #hub to: #pc1.

    self assert: (hello isAddressedTo: pc1).
    self assert: (hello isOriginatingFrom: mac).

    mac send: hello.
    self deny: (pc1 hasReceived: hello).
    self assert: (firstLink isTransmitting: hello).

    firstLink transmit: hello.
    self deny: (pc1 hasReceived: hello).
    self assert: (secondLink isTransmitting: hello).

    secondLink transmit: hello.
    self assert: (pc1 hasReceived: hello).
```

If you run this test, you will see that it fails because of the `notYetImplemented` message we left earlier in `receive:from:`; it’s time to fix that! When a node receives a packet, but it is not the recipient, it should forward the packet:

```
KANetworkNode >> receive: aPacket from: aLink
    aPacket destinationAddress = address
        ifTrue: [
            self consume: aPacket.
            arrivedPackets add: aPacket ]
        ifFalse: [ self forward: aPacket from: aLink ]
```

Now we need to implement packet forwarding, but there is a trap…

An easy solution would be to simply `send:` the packet again: the hub would send the packet to all its connected nodes, one of which happens to be `pc1`, the recipient, so all is good!

*Wrong!*

The packet would be also sent to other nodes than the recipient. What would those nodes do when they receive a packet not addressed to them? Forward it. Where? To all their neighbours, which would forward it again… and again… and again… so when would the forwarding stop?

We should improve our test to demonstrate that hubs don’t send the same packet back to the originating node:

```
KANetworkTest >> testSendViaHub
    | hello mac pc1 firstLink secondLink |
    hello := KANetworkPacket from: #mac to: #pc1 payload: 'Hello!'.
    mac := net nodeAt: #mac.
    pc1 := net nodeAt: #pc1.
    firstLink := net linkFrom: #mac to: #hub.
    secondLink := net linkFrom: #hub to: #pc1.
    backLink := net linkFrom: #hub to: #mac.

    self assert: (hello isAddressedTo: pc1).
    self assert: (hello isOriginatingFrom: mac).

    mac send: hello.
    self deny: (pc1 hasReceived: hello).
    self assert: (firstLink isTransmitting: hello).

    firstLink transmit: hello.
    self deny: (pc1 hasReceived: hello).
    self deny: (backLink isTransmitting: hello).
    self assert: (secondLink isTransmitting: hello).

    secondLink transmit: hello.
    self assert: (pc1 hasReceived: hello).
```

The link backLink monitors whether hub sends the hello packet back to mac when it forwards the message upon its receipt.

To get this test to pass we need hubs to behave differently from nodes. In reality hubs work at the lower layers of the OSI model, but our simplified model does not have that level of detail. We can approximate this by saying that upon reception of a packet addressed to another node, a hub should forward the packet, but a normal node should just ignore it.

Let's first define an empty forward:from: method for nodes, then add a new class for hubs, which will be modeled as nodes with an actual implementation of forwarding:

```
KANetworkNode >> forward: aPacket from: arrivalLink
    "Do nothing. Normal nodes do not route packets."
```

## 2.13 Introducing a new kind of node

Now we define the class KANetworkHub that will be the recipient of hub specific behavior:

```
KANetworkNode << #KANetworkHub
    package: 'NetworkSimulator-Core'
```

We can also create this class through the *Refactorings* menu option on the KANetworkNode class. To do this we open the menu, select *New subclass* and then enter the new subclass name.

Now it's time to implement forward:from: for our KANetworkHub. As the hub has no routing information we're forced to rely on a flood routing algorithm. But... remember the constraint as outlined in our test above: we do not want our hub forwarding the packet back to the originating node. We suggest taking advantage of the message linksTowards:do: that performs an action for all given links to one address:

```
KANetworkHub >> forward: aPacket from: arrivalLink
   ... Your code ...
```

Now we can use a proper hub in our test, replacing the relevant line in buildNetwork, and then check that the testSendViaHub unit test passes.

```
    hub := KANetworkHub withAddress: #hub.
```

You have now a nice basis for simulating networks. But we can do more! In the following section we will present some possible extensions.

## 2.14 Other examples of specialized nodes

In this section we will present some extensions to our network simulation package that will allow us to support new scenarios. We will propose some tasks to make sure that the extensions are fully working. In this section we do not define tests - but we *strongly* encourage you to start to write your own tests. At this point in the book you should be comfortable writing your own tests, and should see their value for improving your development process. You should also consider writing your tests *first*, before you start implementing the examples below, following the practice of Test Driven Development. So please take this opportunity to practice.

### Workstations counting received packets

We would like to know how many packets specific nodes are receiving. In particular when a *workstation* node consumes a packet, it increments a packet counter.

> **Important** Can you write a test for the requirement above? Do you think you could use it to drive the development of a KANetworkWorkstation class, whose instances implement the above behaviour? What other behaviours should the instances exhibit? If you can, compare your solution to the one below. If you don't feel comfortable writing your test first yet, don't worry; just continue reading and implement the solution below.

Let's start by subclassing KANetworkNode:

```
KANetworkNode << #KANetworkWorkstation
    slots: {#receivedCount};
    category: 'NetworkSimulator-Nodes'
```

We need to initialize the receivedCount instance variable for workstations. Redefining initialize should do the trick but, because we inherit instance variables from the superclass (address, outgoingLinks, arrivedPackets and loopback), it's really important to send the super initialize message. If we didn't, some of the those instance variables would not receive their default values (address is initialized in the KANetworkNode >> withAddress: constructor, which our workstations will also use).

```
KANetworkWorkstation >> initialize
    super initialize.
    receivedCount := 0
```

Now we can override consume: in KANetworkWorkstation:

```
KANetworkWorkstation >> consume: aPacket
    receivedCount := receivedCount + 1
```

We should also:

- define accessors and the `printOn:` method for debugging.
- define a test (or multiple tests) for the behavior of workstation nodes.

## Printers accumulating printouts

When a printer consumes a packet, it prints it. We can model the output tray of a printer as a list where packet payloads get stored, and the supply tray as the number of blank sheets remaining in the printer.

> **Important** Again, if you feel comfortable writing a test first based on these requirements, feel free to do so before reading any further and then compare your solution to the one below.

The implementation is very similar to what we did for workstations: we subclass `KANetworkNode` and redefine the `consume:` method:

```
KANetworkNode << #KANetworkPrinter
    slots: { #supply . #output};
    package: 'NetworkSimulator-Nodes'
```

```
KANetworkPrinter >> consume: aPacket
    supply > 0 ifFalse: [ ^ self "no paper, do nothing" ].

    supply := supply - 1.
    output add: aPacket payload
```

Initialization is a bit different, though; since the standard `initialize` method has no argument, the only sensible initial value for the `supply` instance variable is zero:

```
KANetworkPrinter >> initialize
    super initialize.
    supply := 0.
    tray := OrderedCollection new
```

We therefore need a way to pass the initial supply of paper available to a fresh instance:

```
KANetworkPrinter >> resupply: paperSheets
    supply := supply + paperSheets
```

For convenience, we can provide an extended constructor to create printers with a non-empty supply in one message:

```
KANetworkPrinter class >> withAddress: anAddress initialSupply:
    paperSheets
    ^ (self withAddress: anAddress)
        resupply: paperSheets;
        yourself
```

To finish up we can:

- define some useful accessors and our good friend the `printOn:` method for debugging purpose
- define some test methods for the behavior of printer nodes (if you haven't done so already)
- extend the behaviour further. Are there more messages that a printer could usefully respond to?

### Servers that answer a request

Let's make a server which responds to a packet by taking its payload, converting it to uppercase, and then sends it back to the server which sent it.

> **Important** Again, you can stop here and try to drive the implementation our new server with tests if you're comfortable doing that. Otherwise, read on.

This is another subclass of `KANetworkNode` which redefines the `consume:` method, but this time the node is stateless, so we have no initialization or accessor methods to write:

```
KANetworkNode << #KANetworkEchoServer
    package: 'NetworkSimulator-Nodes'
```

```
KANetworkEchoServer >> consume: aPacket
    | response |
    response := aPacket payload asUppercase.
    self send: (KANetworkPacket
        from: self address
        to: aPacket sourceAddress
        payload: response)
```

Define a test for the behavior of server nodes.

## 2.15 Conclusion

In this chapter we built a little network simulation system, step by step. We showed the benefit of good protocol decompositions.

As a further extension, we suggest modeling a more realistic network with cycles, as shown in Figure 2-7. Making this work properly will require replacing our primitive hubs with real routers, and flood routing with more realistic routing algorithms.

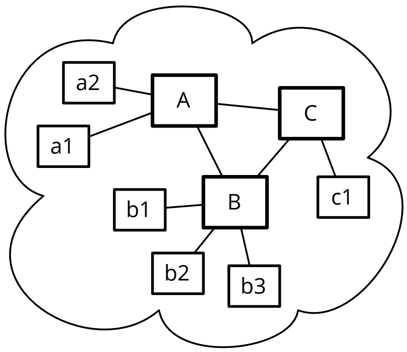


**Figure 2-7** A possible extension: a more realistic network with a cycle between three router nodes.

To start you off, here is a possible setup for a new family of tests:

```
KARoutingNetworkTest >> buildNetwork
    | routers |
    net := KANetwork new.

    routers := #(A B C) collect:
        [ :each | KANetworkHub withAddress: each ].
    net connect: routers first to: routers second.
    net connect: routers second to: routers third.
    net connect: routers third to: routers first.

    #(a1 a2) do: [ :addr |
        net connect: routers first
            to: (KANetworkNode withAddress: addr) ].
    #(b1 b2 b3) do: [ :addr |
        net connect: routers second
            to: (KANetworkNode withAddress: addr) ].
    net connect: routers third
        to: (KANetworkNode withAddress: #c1)
```