An Introduction to XForms

November 27, 2018

Steven Pemberton (/authors/steven-pemberton/)

Steven Pemberton gives us an introduction to XForms with several examples of how useful it is.

Contents

Introduction

XForms is a declarative XML-based programming language. It is a W3C standard, and in use in companies around the world. Experience has shown that using it reduces the time to produce applications by an order of magnitude (what would have taken a week, now takes a morning).

This article is the first of a planned series on XForms, how it works, and how to use it; this first article gives some background and motivation.

The first municipal computer

As I write, it is thirty years since the first country outside of North America (the Netherlands) connected to the open internet, thus making the internet truly international. I would contend that we still think of the internet as 'new', and yet on that day thirty years ago, it was only thirty years before that, that the first computer was installed in a municipality (Norwich, UK surprisingly enough), thus making computers for the first time truly public.

This photo shows just one of 21 cabinets making up that computer. (Image from the Norfolk Record Office, reference NRO, ACC 2005/170.)

At that time, computers were so expensive (of the order of millions, scaled to modern values) that nearly no one bought them, but leased them instead. Even to rent time on a computer would cost you around $1000 per hour (unscaled): more than the annual salary of a programmer!

When you leased a computer in those days, as a sort of sweetener, you would get a number of programmers for free as part of the deal. This was useful, because computers were all different then and so it was good to have someone in-house when the computer arrived who already knew how to program it, and who could train others.

Since the computer's time was so expensive, typically a programmer would:

- write the program on paper,
- copy it out onto special paper,

- give it to a typist, who would type it out,
- who would then give the result to another typist who would then type it out again to check it.

All this extra work was justified because it was *much* cheaper to let 3 people check the program than to let the computer discover the errors.

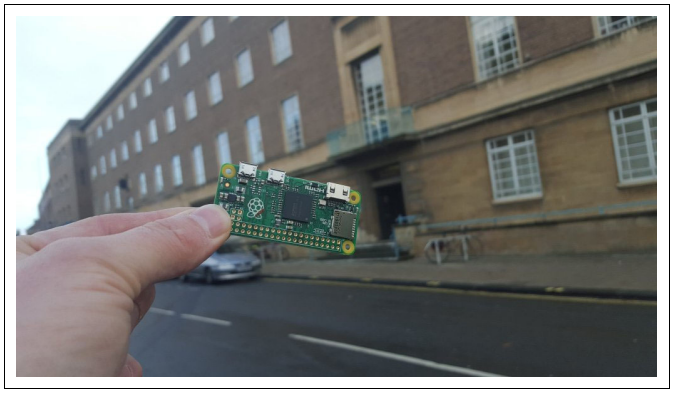
Put another way: *compared to the cost of a computer, a programmer was almost free.*

Programming languages

The first programming languages were designed in those years, and they were designed with this economic relationship between computer and programmer in mind. Since it was much cheaper to let the programmer spend lots of time producing a program than to let the computer do some of the work, programming languages were designed so that you had to tell the computer exactly what to do, in its terms, and not in terms of what you wanted to achieve.

Modern computers

The Raspberry Pi Zero was another milestone in computing, because it was the first computer that was so cheap, it was given away for free on the cover of a magazine. I am grateful to Richard Hancock of Norwich for this photo of the Pi Zero at the same location as the Elliott computer above.



The Elliott ran for about a decade, 24 hours a day: how long would it take the Raspberry Pi Zero to duplicate that amount of computing? Five minutes! The Raspberry Pi is about *one million* times faster...

But it is not only one million times faster: it is also one millionth the price. A factor of a million million. A terabyte is a million million bytes: nowadays we talk in terms of very large numbers, that we can't really grasp in human terms. How long is a million million seconds? 30,000 years... A million million is a *really* big number...

Funnily enough, a million million times improvement is about what you would expect from Moore's Law over 58 years (even though Moore's Law didn't get proposed for the first time until 1965). Except: the Raspberry Pi is two million times *smaller* as well, so it is *much* better than even that. This is partially explainable by the fact that when new technologies are introduced, typically they are not priced based on what they cost to produce, but in terms of the cost of whatever they are replacing. For instance Edison when he first introduced electric light, priced electricity to match what it would otherwise have cost you to produce the same amount of light by other means, the advantage of electricity not being the price, but the ease of use of being able to instantly switch on the light, and the added safety of not having to have a flame.

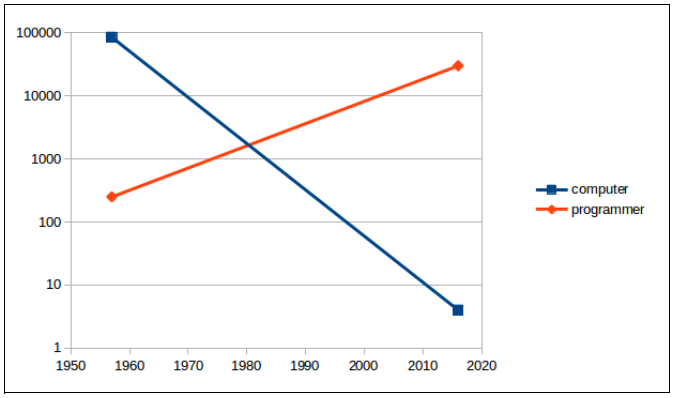
Similarly early computers were priced in terms of how much it would have cost you otherwise to do the same calculations using human power, with the added advantage of more trustworthy results, and greater stamina, amongst others.

Moore's Switch

It happened slowly, almost unnoticed: the cost of programmers grew inexorably, while at the same time the cost of computers plummeted. As the Raspberry Pi so clearly illustrates, nowadays we have reached the exact opposite position: *compared to the cost of a programmer, a computer is almost free.*

And yet, we are still programming in programming languages that are direct descendants of the languages designed in the 1950s!

We are still telling the computers what to do rather than what we want.



Declarative programming

Declarative programming is a new way of programming that describes *what* you want to achieve, but not exactly *how* to achieve it.

To illustrate the difference between a declarative and a procedural definition, take the case of square roots. We learn in school what numbers are, and how to add, subtract, multiply and divide. However, when we get to square roots, we are only told: *The square root of a number n is the number r such that r × r = n.* This is a *declarative* definition. It tells you what something is, it tells you how to recognise it, but it *doesn't* tell you how to calculate it. Most people know what a square root is, but few people leave school knowing how to calculate one.

Now take a look at a procedural definition of square root:

```
function f a:
{
   x ← a
   x' ← (a + 1) ÷ 2
   eps ← 1.19209290e-07
   while abs(x - x') > eps × x:
   {
      x ← x'
      x' ← ((a ÷ x') + x') ÷ 2
   }
   return x'
}
```

Probably if I hadn't told you that this was the code for square root, you wouldn't have guessed, nor have been able to easily work it out. It nicely illustrates the distance between problem statement and solution imposed by procedural programming: you have to translate from one to the other, with only tenuous links between the two. It is one of the reasons why documentation is so essential in procedural programming.

But even knowing that the code is for square root, you are still left with questions. Under what conditions does it work? How does it do it? What is the theory behind it? Is it correct? Can I prove it? Under what conditions may I replace it, or a part of it with something else?

Even if you do know the theory behind it, it is still difficult in this case to see that the theory has been applied, because several steps have been optimised (the loop has been unrolled once, and constant expressions have been evaluated), consequently making the relationship with the theory less transparent.

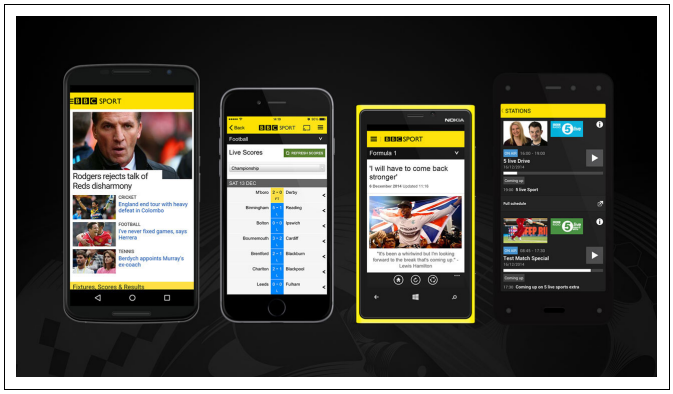
To summarise the advantage of the declarative approach over the procedural one:

- it describes the problem, and the shape of the solution;
- it is (much) shorter;
- it is easier to understand;
- it is independent of implementation;
- it is less likely to contain errors;
- it is easier to see that it is correct;

- it is tractable.

XForms

XForms is a declarative markup for defining applications.



It is a W3C standard, and in worldwide use, for instance by the Dutch Weather Service, KNMI, many Dutch and UK government websites, the BBC, the US Department of Motor Vehicles, the British National Health Service, and many others. Experience with XForms has show that it allows you to produce applications in much less time than with traditional procedural methods, typically a tenth of the time.

For example, a company that makes one-off *big* machines (so big that you walk in) decided to run a pilot one year with XForms. Creating a user-interface for their machines was very demanding, they knew that they normally needed 5 years and 30 people to do it. With XForms this became 1 year with 10 people, including learning time. In other words, assuming one person costs the company 100k a year, it has gone from a 15M cost to a 1M cost. They have saved 14 million! (And 4 years).

Another example is a British insurance company. A manager had decided to evaluate whether to use XForms, and asked two programmers to come back in two days' time with an estimate of how long it would take them to produce the next application, one using their normal procedural approach, the other using XForms. Two days later, the procedural programmer asked for another 30 days to work out how long it would take; the XForms person meanwhile had already programmed it.

The current XForms poster child is at the British National Health Service. They started a project for a distributed national health records system: it involved 70 people, and it cost many billions of pounds. The hardware costs alone of the resulting system were an immense £5 per patient; however it failed as non-performant. One person working alone then created a system using XForms (https://seveninformatics.com/cityehr/), where the hardware costs are 1p per patient, that can run on Raspberry Pi's, and is now running in 5 NHS hospitals.

XForms, the high-level view

XForms 1.0, as the name suggests, was originally designed for online Forms. However, after some experience it was realised that the design had followed HTML too slavishly, and with some slight generalisation, it could be more useful, and so was born XForms 1.1 [1], a Turing-complete, declarative programming language. It now has implementations from Belgium, France, Germany, NL, UK, and the USA, amongst others. The next version, XForms 2.0, is in preparation [2].

In a Wikipedia article on "Form and Content (https://en.wikipedia.org/wiki/Form_and_content)", it says: "The term *form* refers to the work's style, techniques and media used, and how the elements of design are implemented. *Content*, on the other hand, refers to a work's essence, or what is being depicted." This is a nearly-perfect description of XForms, and therefore you can think of the *form* in *XForms* in that sense rather than the earlier sense. XForms applications indeed have two parts:

1. the model, describing the data, and its relationships;
2. the user interface, describing the content, and connecting to the values in the model.

XForms is all about *state* (which means it meshes well with the web REST protocol -- Representational State Transfer). The data used can be internal or come from external sources; you describe the data: properties and relationships; you display any selection of values in the content.

Initially the system is in a state of *stasis*. When a value changes, by whatever means, the system updates related values to bring it back to stasis. This is like spreadsheets, but *much more* general. The result means that programming is *much* easier, since the system does much of the administrative work for you.

An example

To give a very simple example, in order to give a taste of the ease the approach gives, say you've got a position in the world as x and y coordinates, and you want to display the map tile of that location at a certain zoom level. The data is: `x, y, zoom;`

Openstreetmap has a REST interface for getting such a tile:

```
https://openstreetmap.org/<zoom>/<x>/<y>.png
```

The one hiccup is that the coordinate system changes at each level of zoom: as you zoom out, there are half as many tiles in each axis, so there are ¼ as many tiles in total, and the interface indexes tiles, not locations.

So to get a tile, you have to know how big a tile is , and you have to calculate the correct index using this along with the zoom level. So with the data `x, y, zoom` , the calculations are:

```
 maxzoom = 19
tilesize = 8
scale = 2^(maxzoom + tilesize - zoom)
tilex = floor(x/scale)
tiley = floor(y/scale)
url = concat("https://tile.openstreetmap.org/", zoom, "/", tilex, "/", tiley, ".png")
```

That is really all that is needed, modulo syntax, which looks like this:

```
<bind ref="tilex" calculate="floor(../x div ../scale)"/>
```

That's the *form*. Now the *content*:

```
<input ref="x" label="x"/>
<input ref="y" label="y"/>
<input ref="zoom" label="zoom"/>
<output ref="url" mediatype="image/*"/>
```

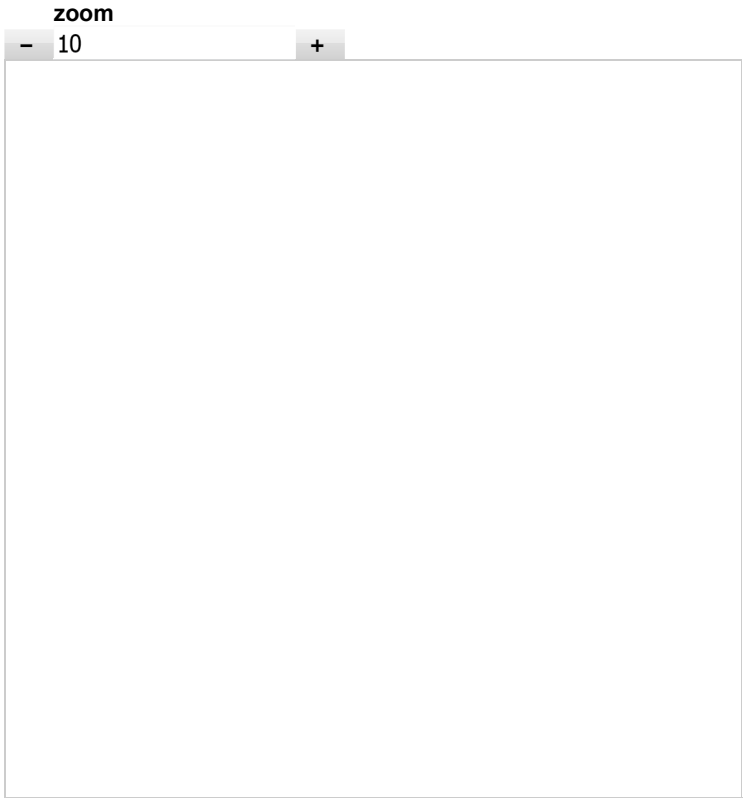
and the tile will be updated each time any of the values change. Here it is working; you can try it out:

zoom
– 10 +
x
← 68955204 →
y
↑ 44117334 ↓
Scale
131072
http://tile.openstreetmap.org/10/526/336.png

Source (https://www.cwi.nl/~steven/forms/examples/xmlcom/maptile.xhtml)

Once you've got this simple version, it is little extra work to turn it into a fully-fledged map application, where you can drag the map

with your mouse, and so on, as this one below. This will be the subject of a future article in the series.



Source (https://www.cwi.nl/~steven/forms/examples/xmlcom/map.xhtml)

XForms, some detail

As mentioned above, XForms has a separation of the data from the user interface, which is similar to how you get separation of style from content with CSS, with similar advantages.

You create *instances* of data and bind properties to the data values:

```
<instance src="sale.xml"/>
<instance>
   <tile xmlns="">
      <x>68955204</x>
      <y>44117334</y>
      <zoom>10</zoom>
   </tile>
</instance>
<bind ref="x|y|zoom" type="integer"/>
```

Properties include types, constraints on values, conditions under which the values are relevant, conditions under which values are optional or required, and how values relate to other values, with calculations.

As was shown above in the map example, whenever a value changes, the related values are automatically updated, in spread-sheet style.

To give an example of how these properties operate, consider the *required* and *relevant* properties. If an address is being filled in, then the value for *state* is only required if the address is for the USA; otherwise it is optional. You specify that as follows:

```
<bind ref="address/state"
      required="../country = 'USA'"
      label="State"/>
```

Controls in the user interface then bind to data nodes, inheriting their properties.

```
<input ref="address/state"/>
```

In this case, the input field for state will be marked specially by the system to indicate that it is required if the country value is set to USA:

Street

City

Postcode

State

Country

Source (https://www.cwi.nl/~steven/forms/examples/xmlcom/address0.xhtml)

On the other hand, if we write:

```
<bind ref="address/state"
      relevant="../country = 'USA'"
      required="true()"
      label="State"/>
```

then `state` is only relevant, and will only be visible, for the USA, but once visible will be required:

Street

City

Postcode

Country

Source (https://www.cwi.nl/~steven/forms/examples/xmlcom/address1.xhtml)

Similarly, a billing address is only relevant if it is different from the delivery address:

```
<bind ref="address[@type='Billing']"
      relevant="../@different=true()"/>
```

Delivery and billing address are the same

Delivery address
Street
City
Country

Billing address
Street
City
Country

Source (https://www.cwi.nl/~steven/forms/examples/xmlcom/address2.xhtml)

Events and Actions

Typically XForms works automatically. However it is possible to hook into the processing model to respond in special ways: *events* announce changes in the state, *actions* effect changes to the state, and they can listen for events in order to respond.

As an example, the event *xforms-ready* announces that the system has initialised (and is at stasis). You could respond to this with the `setvalue` action to record today's date and time:

```
<action ev:event="xforms-ready">
   <setvalue ref="today" value="now()"/>
</action>
```

Other events announce when a value changes, or when it changes validity, relevance, etc.

Other useful actions include inserting and deleting elements and attributes in data, and activating a button (called a *trigger* in XForms). In fact the only way to get a vanilla trigger to do anything is to listen for the activation event, and respond with an action.

```
<trigger label="Restart">
   <action ev:event="DOMActivate">
      <setvalue ref="score" value="0"/>
   </action>
</trigger>
```

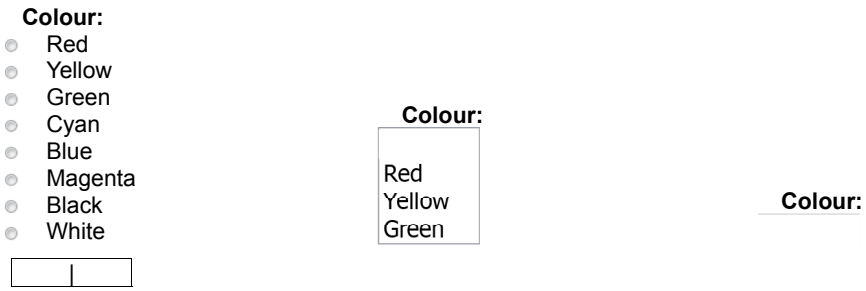
Content

The user-facing part is done with *controls*. These are declarative too: they are designed to be device and modality independent, and describe *what* they do, but not *how* they do it, nor *how* they look.

For instance, the `select1` control selects a value from a list of values. One was used above for selecting the country in an address. It can be implemented visually as a menu, or as radio buttons, and it can be implemented in other modalities as necessary.

```
<select1 ref="colour">
   <label>Colour:</label>
   <item><label>Red</label><value>red</value></item>
   <item><label>Yellow</label><value>yellow</value></item>
   <item><label>Green</label><value>lime</value></item>
   <item><label>Cyan</label><value>aqua</value></item>
   <item><label>Blue</label><value>blue</value></item>
   <item><label>Magenta</label><value>fuchsia</value></item>
   <item><label>Black</label><value>black</value></item>
   <item><label>White</label><value>white</value></item>
</select1>
```

For instance these three are just different visual representations of this control. Since they are all bound to the same value, if you change one, the others will change to match:



Source (https://www.cwi.nl/~steven/forms/examples/xmlcom/controls.xhtml)

What next?

With this brief introduction behind us, future articles will each treat one example application using XForms to demonstrate the techniques used.

How to

The easiest way to try XForms out without having to install anything is to download XSLTForms (https://sourceforge.net/projects/xsltforms/files/latest/download), and unpack it in a directory. Write your XForm using this template, filling in the correct directory, and serve up the result so that it is delivered by your web server with an XML mediatype, for instance by naming the file with a .xml or .xhtml suffix:

```
<?xml-stylesheet href="directory/xsltforms.xsl" type="text/xsl"?>
<html xmlns="https://www.w3.org/1999/xhtml"
xmlns:ev="https://www.w3.org/2001/xml-events">
<head>
  <title></title>
  <style type="text/css">
      add your styling here
  </style>
  <model xmlns="https://www.w3.org/2002/xforms">
    <instance>
      <data xmlns="">
         add your data here
      </data>
    </instance>
  </model>
</head>
<body>
  <group xmlns="https://www.w3.org/2002/xforms">
      add your controls here
  </group>
</body>
</html>
```

References

[1] John M. Boyer (ed.), XForms 1.1, https://www.w3.org/TR/xforms/ (https://www.w3.org/TR/xforms/)

[2] Erik Bruchez, *et al.* (eds.), XForms 2.0 (in preparation) https://www.w3.org/community/xformsusers/wiki/XForms_2.0 (https://www.w3.org/community/xformsusers/wiki/XForms_2.0)