8

Case study: Transforming a legacy GUI to WPF and IronRuby

When I launched a call for contributions to this book from the community to get some real world examples. I asked people to write a short article about what they were doing. As it turns out the contributions I got sent in were a lot bigger than I expected. So I had to do some filtering and decided to go with 2 community contributed chapters in this book. The first one being the previous chapter where we learnt about AutoCAD and building DSL’s with IronRuby.

This is the second chapter that is contributed by a member from the community. This chapter has been contributed by Robert Brotherus from the company Napa in Finland. The company makes software for designing ships. This is a case study on how they are using IronRuby to do some pretty complicated things. From here on out I’ll give the word to Robert and we’ll meet again at the end of this chapter.

8.1 Napa past and present

Our company, www.napa.fi, makes Napa-software for ship design and operation. The development of the first version of Napa software started already in 1979 with few programmers and Fortran77, the cutting-edge language of the day. The exceptional lifespan of the our software –now 30- years and development still going strong – has allowed Napa to acquire over 90% market share in the area of initial ship design at the worlds dockyards but has meanwhile created a significant issue of legacy technologies and architectures, especially regarding the user interface. The original user interface consisted of ASCII-terminal connection to a mainframe computer – the only graphical part of the interface at those days being the support for pen-plotter devices drawing pictures of the ships. Starting from early 1990's, development of graphical user interface started with X-Windows and Motif -technologies, combined with custom widget editor, custom widget-serialization syntax and GUI-code written in custom in-house interpreted language “NapaBasic”. Today over 200 man-years of programmer effort have been invested by 10-20 programmers in producing about 1 million lines of GUI-codes with these technologies.

There have been no new releases of the Motif platform since the 2.1 release over a decade ago, so the platform is essentially dead. Motif lacks many valuable features of modern GUI toolkits and it requires inconvenient X-Windows emulator software to run on Windows, the only platform that our clients use today. Our in-house NapaBasic language lacks many features of modern programming languages like objects, managed memory and extensive out-of-the-box framework libraries. Using an in-house custom language also ties up valuable resources for maintenance of the NapaBasic interpreter (made in Fortran). For any of these technologies one cannot hire new developers that would be already familiar and productive with them. The libraries of documentation, web sites and other support one can get for them is small compared to modern standard alternatives.

These factors amount to a significant hit on productivity for adding new features to Napa software. During the recent years it has become increasingly clear, that for Napa software to survive in the long term, Napa has to move to technologies that are more modern, more feature-rich and more standard.

8.2 Paths to the future

We launched our GUI renewal project at 2007 after months of evaluation of various alternatives. A decision was finally made in the end of 2007 to go ahead with the leading-edge combination of WPF as the widget toolkit, XAML as the widget description syntax and IronRuby (then at very early pre-alpha) as the GUI logic language. The question was then, how to make the transition happen?

One obvious alternative is to rewrite the GUI from scratch. However, like all companies, we face continuous business pressure to develop new features for our software to satisfy the developing needs of our customers. Hence the option of putting our whole GUI-layer team to making the new GUI for few years in a row with no releases of the Napa software coming during that time is not viable for the business. At any point of time most GUI-developers must remain committed to feature development within the legacy platform and only a handful can be allocated to architectural efforts like the GUI renewal.

Given the amount of code and features in the GUI layer, even with optimistically higher productivity we would be looking at 10+ years of work with our small GUI-renewal team before any release of the WPF-Ruby version of Napa. Furthermore the fact that feature-development in the legacy platform is ongoing would mean that the GUI renewal team would be shooting a moving target. Any part of the GUI re-coded with the new techs would require further modifications to keep up with new features added to the legacy platform. This would very likely mean that no amount of time in the world would be enough to produce a feature-identical and not-too-buggy WPF-Ruby version of the Napa GUI: the more we would code, the more we would lag behind.

These reasons prompted us to investigate and later adopt the less obvious, more technically challenging but potentially more feasible approach of legacy transformation (http://www.semdesigns.com/Company/Publications/Legacy%20Transformation.pdf). Legacy transformation is a form of metaprogramming: writing programs that write programs. Metaprogramming might be a familiar term to Ruby-programmers since Ruby (and IronRuby) supports dynamic run-time modification of any program by itself. Legacy transformation involves static metaprogramming: production of the new-tech codebase from the old-tech codebase before runtime. In the context of Napa GUI renewal this means development of parsers that read, analyze and transform the legacy Motif widget definitions to WPF-XAML and existing NapaBasic GUI code to IronRuby. While easy task to define in one sentence, it is far from trivial.

Compared to a rewrite, the transformation strategy has potentially a significantly higher productivity since the code to write for the transformation rules (estimated at 5000 lines) is only a fraction of a percent of the codebase to transform (1+ million lines). The bigger the codebase, the more attractive the transformation path comes compared to rewrite. In a small transform codebase there will be also less bugs than in a big rewrite codebase and any bugs are bound to be more easy to detect since they have effect on thousands of locations in the final codebase. More importantly, feature development within the legacy platform can continue uninterrupted during the development of the transformation rules. All the new features and bug-fixes added will move automatically to the new platform simply by re-running the transformation. This eliminates the problem of shooting a moving target (and related high risk of never reaching the target because of the moving).

While the 5000 to 1 million lines ratio of transformation versus rewrite gives a theoretical 200 x productivity-edge for the transformation, things are not so rosy in practice – otherwise legacy transformation projects would be much more common in the IT-industry than they are now. The big downside and risk in legacy transformation is that writing any single line of transformation code is technically much harder than writing one of the million lines of code in a rewrite-project. Syntaxes of source languages must be mapped to syntaxes on target languages and they rarely match well. Classes and attributes of source object model must be mapped to classes and attributes of target object model and they often match even worse. Framework library calls to the legacy platform have to be mapped to library calls on the target platforms and they are often badly matching. Lots of “glue”, “patching” and “bridging” is needed to make the initially incompatible jigsaw pieces fit together.

In the end we aim to have the transformation fully integrated to our build process so that during a transition period we can produce two fully functional and working applications automatically from the same legacy codebase: one using the legacy technologies and other using WPF and IronRuby. Such a transition period will be followed by a switch where the legacy codebase is retired and the transformed XAML and Ruby will be declared first-class source files. After this the transformation is never run again and the new source files become the target for further evolutionary development by hand editing.

The next part discusses the anatomy of such a transformation.

8.3 Dissecting a transformation

The transformation is staged in two parts, shown by the large orange arrows in the diagram (figure 8.1). The first part handles the transformation of the static parts of the widget definitions and produces XAML-files as its output. This part uses XSLT as the transformation technology, taking advantage of the fact that both its input (Napa Motif-GUI widget definitions in \*.NDS files) and output (XAML) are XML. The majority of rule definitions (as XSLT templates) in this stage handle mapping of Motif primitive widget-classes to WPF primitive widget-classes and Motif widget-properties (“resources” in Motif-terminology) to WPF widget-properties. In many cases there is no 100% one-to-one correspondence between the primitive widget-classes of these different frameworks, then some custom coding of WPF widgets that emulate Motif-widgets and/or properties was needed. These are included in the yellow “new code” boxes in the diagram.

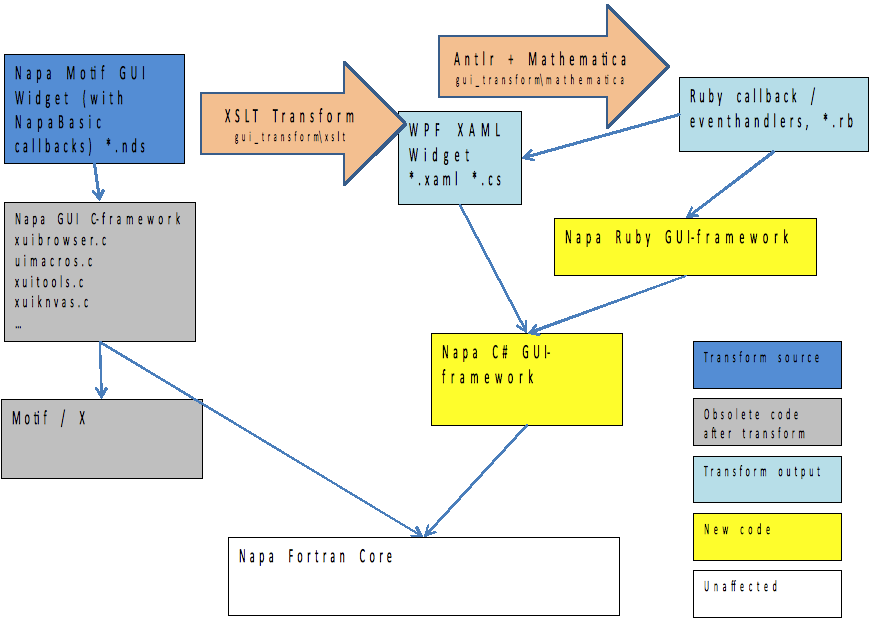


Figure 8.1: The transformation process

The second and significantly more challenging part of the transformation is the transform of NapaBasic GUI logic code to (Iron)Ruby code. This part starts with a parser of NapaBasic that produces AST’s (Abstract Syntax Trees). These AST’s are transformed to (Iron)Ruby AST’s and finally (Iron)Ruby code is generated from these AST’s. The NapaBasic parser has been developed as a custom grammar for the ANTLR 3 parser generator. The AST transform was done with Wolfram Mathematica software. Mathematica supports functional pattern-based rule-programming that is particularly suitable in defining the complex rules of mapping language constructs and library calls of the Motif GUI to language constructs and library calls of the Ruby/.Net/WPF platform. The AST transform is by far the most challenging part of the GUI renewal project. While changing of syntactic forms of one language to another is straightforward in itself, this part of transformation includes much more. For one all the mappings of Motif widget-classes to WPF classes and Motif widget attributes to WPF properties must be done in this stage as well since new widgets are created in the code and their attributes are manipulated dynamically.

8.4 Choosing IronRuby

XAML and Ruby on their own are standard and well-supported technologies and most likely IronRuby will soon be that as well. However, the combination of XAML and IronRuby for GUI-development is much less standard and we are unsure whether that status is going to improve. The standard method of compiling XAML to static CLR-classes in Visual Studio is not a good match for DLR-based Ruby. The interoperability issues between the languages and their different type-systems carries a price. The relative immaturity of IronRuby creates a problem of bugs and lacking features in the language and poor tool/debugging support. Therefore the question arises why choose IronRuby as the language for GUI logic instead of C#, the more standard language for WPF event handler coding? The answer is not obvious and careful weighing of the alternatives was done before proceeding with (Iron)Ruby.

The argument for IronRuby (and against C#) in the end was following. First, we estimated that it would provide lower learning curve and hence higher productivity for our many marine-engineer developers. Second, NapaBasic is a dynamically typed language and that translates more naturally to another dynamically typed language instead of the statically typed C#. Since very little type information is present in the source language, it is difficult to generate enough of it to the target language to satisfy the C# compiler. Third, the power and flexibility of Ruby's various meta-programming capabilities (like addition of methods and local variables to object at runtime) help to create part of the “glue”, “patching” and “bridging” that are needed to make the transform between the different technologies work. Finally, any issues that are raised because of the immaturity of IronRuby, we were optimistically hoping to vanish before we would reach production state in the transformation. From the developments seen in IronRuby after our decision it seems that our optimism has not been in vain). Listing 8.1 is an example of IronRuby-code resulting from the transformation.

Listing 8.1: A conversion result

...

# NB: @if com='Delete' then

if ( com == "Delete" )

# NB: @if rsize(ar)<10 @end

if ( ar.length < 10 )

return

end

# NB: @for vidx=2,8

for vidx in 2..8

vnr = ar[vidx].to\_f # NB: @vnr=value(ar(vidx+1))

# NB: @if vnr>0 then

if ( vnr > 0 )

str = ( "!VIEW " + vnr.to\_s ) # NB: @str=cnc('!VIEW ',fmt(vnr))

MN.command(str) # NB: @mn.command(str)

drawnr = GR.windownr # NB: @drawnr=gr.windownr()

GR.closedrawable(drawnr) # NB: @gr.closedrawable(drawnr)

else

return # NB: @end

end

end

return # NB: @end

elsif ( com == "Projection" )

...

The transform preserves the original NapaBasic as comments starting with “# NB :” in the transformed code. This is to help our developers (familiar with NapaBasic and Motif widget-attributes) to move to Ruby (and WPF widget attributes). Many statements like mn.command(str) go through the transformation relatively unmodified. These are calls to functionality in the Napa Fortran core and similar calls will remain in the Ruby/Dotnet platform. In that platform such calls will be routed at runtime though IronRuby-C# interoperability to C# and finally to Fortran core (as a native dll) via Dotnet's Platform Invoke (P/Invoke) and supplementary C#-Fortran interoperability code generated with SWIG (http://www.swig.org).

The transformation occasionally produces unnecessary pairs of parenthesis. This is a feature of our AST-Ruby-Serialization code that inserts parenthesis in a bit of a paranoid fashion to avoid lacking parenthesis where they would be really needed. Removal of unnecessary parenthesis and other similar cleanup work can be done later by hand editing when the transformation output files have been declared first-class source files. Of course there are trade-offs that need to be made as well as some though choices.

8.5 XAML and IronRuby: trade-offs and choices

The marriage of XAML and IronRuby has had its good and bad times in our transformation project, but to all issues encountered, reasonable solutions have been found so far.

XAML can be used in a static or dynamic way. The dynamic nature of IronRuby would suggest a more natural relationship with dynamic XAML. For example in the same way as IronRuby allows us to change and reload our GUI-codes without recompiling the application, dynamic XAML would allow us to change the GUI-layouts without recompiling.

However, the Visual Studio 2008 standard way to process XAML is static: for each XAML file, a corresponding class (named by the xaml:Class attribute) is created. Instances of that class are instances of the widget described by the XAML. If the top-level element of the XAML is, say, Window, then the constructed class will have Window as its base class. The class serves as a natural placeholder for event-handler methods of the widget.

Using the target class name as the XML-element tag can make references to other XAML-defined widget-classes. For example here MN\_PLTWIN (Listing 8.2) widget is referring (taking as its part) another XAML-defined widget MN\_STATUSBAR.

Listing 8.2: Some widgets defined in XAML

<!-- The PLTWIN Widget -->

<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:wpf="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xaml:Class="Napa.Gui.MNWS.MN\_PLTWIN" Name="PltWin">

<DockPanel LastChildFill="True" Name="PltWn">

<Menu wpf:DockPanel.Dock="Top" Name="MB">

...

</Menu>

<MN\_STATUSBAR xmlns="clr-namespace:Napa.Gui.MNWS" Name="StatusBar"

DockPanel.Dock="Bottom"/>

…

</DockPanel>

</Window>

<!-- The MN\_STATUSBAR -->

<FormPanel xmlns="clr-namespace:Napa.Gui.MotifSupport;assembly=CommonGui"

xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:m="clr-namespace:Napa.Gui.MotifSupport;assembly=CommonGui"

xaml:Class="Napa.Gui.MNWS.MN\_STATUSBAR"

Name="StatusBar">

<InputField\_ProgressBar xmlns="clr-namespace:Napa.Gui.MNWS.MN\_STATUSBAR\_Parts"

Name="ProgressBar" m:FormPanel.AttachTop="FORM" ... />

...

</FormPanel>

In the dynamic XAML alternative, System.Windows.Markup.XamlReader is used to load XAML from a file at runtime. No new classes are created in this case: the resulting widget-tree root widget is an instance of the class corresponding to the root XML-element name. Without classes, referencing from XAML to other XAML-widgets is no longer possible with the natural <MN\_STATUSBAR ... /> syntax. Any attempts to remedy the situation with IronRuby code fail since IronRuby classes are not static CLR-classes expected by XAML-reader but rather dynamic DLR-classes which XAML-reader (or the rest of the .NET 3.5 framework) has no clue about. Furthermore, since there is no new class involved, there is no natural placeholder for the event handler code of the widget.

Both of these issues can be circumvented if one is willing to pay the price and write the plumbing code for a non-standard solution. We did consider circumventing the XAML-reference issue by custom C#-coded “XAMLWidget”-class that would work like <my:XAMLWidget file=”MyCustomWidget.XAML”/> in the client XAML. This would be inconvenient for setting attributes of the custom widget since XML-attributes of a XAML-element must correspond to (static) CLR-properties of the corresponding class. So even if MyCustomWidget has an attribute Background one cannot use the syntax <my:XAMLWidget file=”MyCustomWidget.XAML” Background="Red"/> but instead a more convoluted syntax like <my:XAMLWidget file=”MyCustomWidget.XAML” Attributes="Background: Red; Orientation: Horizontal;"/>.

We also experimented with placing the event handler code in the dynamically loaded XAML-objects (by lack of the class for the widget). IronRuby allows this with the Ruby object-extension syntax as shown in listing 8.3.

Listing 8.3: Adding event handlers to XAML widgets

class << my\_widget\_object

def some\_new\_method

...

end

...

end

However, this approach would produce less standard and obvious code and require special plumbing code to connect the created widget objects with their handler code. The code must be executed for each new widget instance created.

Widget-to-Widget references play a big role in Napa Motif-GUI, so solution that addresses them in a clean and standard way was preferred in the end. This meant sticking to the standard static XAML-to-CLR-class compilation of VS2008 and living with the compromises it requires. It remains to be seen if .NET 4.0 or some later version will allow more “dynamic XAML” with dynamic references and attributes. But we are not counting on that.

Another issue with XAML was its limitations in regards to class inheritance. XAML-defined classes can have any C#-defined widget class as its base class (signified by its root element tag) and C#-defined widget class can derive from a XAML-class. However, a XAML-class cannot have another XAML-defined class as base class. Mostly this makes sense: If you make a XAML-defined panel with some arrangement of buttons as content and derive another widget from that and put some other arrangement of buttons to that, what should be the relation of the two arrangements of buttons? There is no obvious answer to this and hence MS has decided to forbid XAML-inheritance completely. However, there are some special cases where inheritance might have a more clear meaning, like inheriting from a menu and adding more menu items in the descendant class. Napa-Motif uses this kind of inheritance, especially with menus, leading to XAML that does not compile with VS. We had to do quite a lot of refactoring to get the existing codebase to remove widget-definition inheritances to make them transformable to XAML.

A more radical option we considered at some point was to not use XAML at all and use instead a pure IronRuby description of the GUI. While avoiding some of the problems above and being attractive in some other ways, we decided against that option in the end. The reasons for this were:

1. The lack of a standard and supported way of describing WPF-GUI declaratively in IronRuby.
2. The availability of graphical GUI-designer support for XAML.
3. The easier enforcement of separation of GUI-layouts and GUI-code with XAML + Ruby and the hope for better support for XAML + Ruby in future .NET-framework releases.

Now we know how got to our mix of technologies, once that was set we started focusing on issues that arose from using Ruby as a language.

8.6 Ruby issues and solutions

IronRuby is a modern, powerful, feature-rich multi-paradigm language compared to NapaBasic. One would hence assume that for any construct of NapaBasic one would easily find a suitable feature to map to IronRuby and that such a mapping would only require a subset of IronRuby features. While the latter is true, unfortunately the former assumption turns out to be wrong: Ruby as an idealistic modern language actually omits many features that have been found problematic and avoidable. While good for enforcing good style in new code, this creates headaches for defining mapping of code in old languages like NapaBasic.

For example, NapaBasic has the concept of goto-statements and labels to jump to with goto. C# has labels and gotos as well (though their usage is understandably discouraged) but Ruby does not. It is fine not to have gotos in the language when one is writing new code, but we faced the issue as to what kind of Ruby-constructs to produce for the existing gotos. Our solution is two-fold. First, we analyze common patterns of goto-usage and try to map such special cases to specific ruby-constructs. For example some gotos can be detected to be used to form a loop and can be mapped to a while-construct in ruby. Another common example is usage of a goto to jump to an error handler block at the end of a method. In most cases these could be automatically mapped to begin...rescue...end constructs in Ruby. After defining mapping rules for such common cases of goto-usage, we expect to have about 10% of the original gotos left. These “wild” cases must be eventually factored out from the sources by hand to allow transformation to proceed.

NapaBasic also has the concept of subroutines in the style of “Commodore64 Basic” GOSUB / RETURN constructs. These too produce potentially unmappable structures to Ruby. Luckily most of the cases in the codebase seem to conform to a form that allows the subroutine code block to be transformed to a Ruby method and GOSUB to a call of that method. This leaves the issue of local variable scope since NapaBasic subroutines share the scope of the main program but Ruby methods have different variable scopes. Our current solution to this is to detect in the mapping variables that are shared between main and subroutine and convert them to instance-variables that are shareable between methods.

NapaBasic, again like C#, supports out- and ref-parameters to functions in addition to normal in-parameters. We have not found a general solution to this problem that would be as clean as we would like it to be. Our current best alternative involves wrapping ref/out-parameters to some kind of transport-objects and generating code in the transformation to load and unload such transporters as shown in listing 8.4.

Listing 8.4: Simulating out and ref parameters

class Ref < Struct.new(:value); end

def calculate(b) # b is defined as a ref-parameter in NapaBasic

# inside the method block, replace all references to symbol b to b.value

puts b.value # use b value

b.value = 7 # "assign ref value"

end

a = 5

# original: calculate(a)

a\_ref=Ref.new(a)

calculate(a\_ref) # replace parameter in a call with new ref

a = a\_ref.value # generate setting of original val

puts a # prints 7

Now that we’ve found acceptable solutions for the hurdles the language construct mismatches we’re moving on to mapping attributes.

8.7 Mapping attributes

The bread and butter of GUI-logic code is the reading and writing of various GUI widget properties. Hence the bread and butter of Motif-WPF GUI transformation is the logic for mapping Motif-widget property reading and writing statements of NapaBasic to corresponding WPF-statements in IronRuby. In many cases the mapping is straightforward. For example:

NapaBasic: @Mtf.SetResource(acheckbox,'selected','True')

IronRuby: acheckbox.is\_checked = true

Note how NapaBasic is not object-oriented: acheckbox variable in the NapaBasic statement is a widget handle (akin to HWINDOW in windows GDI) and can be used only as a parameter for a global static function Mtf.SetResource (widget properties in Motif are called resources.) In the transformation we map widget handles to WPF widget references and are hence able to set WPF properties with a more clean OO-syntax.

Note also the mapping of the property value from string ('True') in NapaBasic to boolean in Ruby. NapaBasic and Motif do not have the concept of true boolean types, so strings and integers are used instead there. In some other cases the difference in types is even bigger. For example colors in Motif-widgets are simply strings whereas in WPF they are instances of the struct System.Windows.Media.Color. Furthermore, WPF-colors are not directly properties of widgets but of Brushes that can be properties of widgets. So we have for example:

NapaBasic: @Mtf.SetResource(btn,'background',c(i))

IronRuby: btn.background=Wpf::SolidColorBrush.new(c[i-1].to\_color)

The array-variable c contains color names. In Motif-side they are directly usable as widget properties. On the Ruby-side, they are converted to System.Windows.Media.Color instances with the to\_color() method that we have monkey-patched to String-class. Note also in the example the mapping of NapaBasics 1-based arrays to Ruby/.NET 0-based arrays and our usage of module Wpf as a convenient container for classes of all Wpf-namespaces.

To give some taste of the definition of the transformation, in listing 8.5 there are parts of our Mathematica-code (total of ~2000 lines currently) that participate in the previous mapping.

Listing 8.5: Some transformation code

Transform @ nb`UiCall["set"|"setresource", wid\_, pars\_\_] := (

VariableType[wid] = "widget";

SetResource[wid, pars]

);

SetResource[wid\_, attr\_String, value\_, remain\_\_\_] := Sequence[

nb`Assignment[ ToWidgetRef[wid] ~ nb`Dot ~ nb`Call[ToLowerCase[attr]], value],

SetResource[wid, remain]

];

Transform @ nb`Assignment[wid\_ ~ nb`Dot ~ nb`Call["background"], val\_ ] :=

nb`Assignment[wid ~ nb`Dot ~ nb`Call["background"], ToWpfBrush[val] ]

// Transform;

There are many attribute-mappings where the corresponding Motif- and WPF-concepts are so far apart from each other that it is not plausible to write a static mapping like in the examples above. For example, Motif-NapaBasic supports resource children[n] that returns array of child-widget names for some widget:

Napabasic: @ui.getarr(${id},'children[n]',c)

This information is available in WPF, but there is no property that would directly or even semi-directly correspond to this Motif-property. To produce the results it requires a bit more logic that can be expressed in one line and furthermore the logic depends on the particular WPF-class in question. It is not desirable to generate duplicate instances of this logic statically to each call location, so here we do dynamic mapping instead. Statically we generate only a method-call to a custom method:

IronRuby: c = self.children\_names

We introduce and implement this method with monkey patching to all WPF FrameworkElement classes (WPF widget base class) and as needed in different specific WPF-classes as shown in listing 8.6.

Listing 8.6: Monkey-patching WPF classes

class Wpf::ContentControl

def children\_names

...

end

end

class Wpf::ItemsControl

def children\_names

...

end

end

...

When the mappings of attributes have been defined we still need to tackle the problem of wiring up events.

8.8 Wiring events

For the transformed GUI-logic code to respond to GUI interaction, the corresponding methods must be connected to WPF-events of widgets. Current IronRuby supports more than one syntax for connecting .NET-events. For historical reasons we use currently the syntax that has been available from early alpha versions: if widget w has event named click, one can connect a block of ruby code to the event with syntax: w.click { |sender, args| … }. Listing 8.7 shows how we support “configuration by convention” -style (like Ruby On Rails) of connection of methods with name <eventname>\_handler to corresponding events.

Listing 8.7: Example of auto-wired event

class Napa::Gui::MNWS::MN\_PLTWIN\_Parts::Button\_Ok

def click\_handler(args)

...

end

end

In this example, the click\_handler-methods would be automatically wired to click-event of the button at runtime upon the construction of the widget. Listing 8.8 has the framework code that connects the event.

Listing 8.8: The event connector

# Connect Ruby event-handlers / methods to the events in WPF/XAML.

def wire\_events

self.methods.each do |m|

if m =~ /((\w+)\_handler)/

handler\_method = $1; event\_name = $2

connect\_event(event\_name, handler\_method)

end

end

end

def connect\_event(event, handler)

if self.respond\_to?(event)

send(event) do | sender, args | # attach event to this block

begin

self.send(handler, args) # call the handler method

rescue Exception

puts "failed: #{self.class.name}.#{handler}()"

puts "GUI event '#{event}' exception intercepted:"

puts $!

$@.each { |i| puts i }

end

end

end

end

This simplifies our transformation rules since we don’t have to generate the code to connect the events for every widget and every event handler but simply provide properly named method names. Also we hop this will make code more clear and that coding new widgets will be more productive. When we solved the problem of the event handlers there is one more thing we needed to look at and that is constructors.

8.9 Constructors

Monkey-patching methods to WPF-widgets from IronRuby has the obvious limitation that the methods are only visible when called from Ruby – not when the WPF-widgets are constructed from C#. This can create confusing situations regarding eg. constructor-calls. As you can see in listing 8.9 we can add a Ruby-style constructor-method for some transformed WPF-class. This initialize method is then called when the widget is constructed from Ruby.

Listing 8.9: Defining an initialize method in Ruby

class Napa::Gui::MNWS::MN\_PLTWIN\_Parts::Button\_Ok

def initialize

puts "Button\_Ok constructor"

end

end

include Napa::Gui::MNWS::MN\_PLTWIN\_Parts

w = Button\_Ok.new

# prints: Button\_Ok constructor

However there is a problem when we would be calling this code from a C# defined class as illustrated by listing 8.10.

Listing 8.10: The Button\_Ok widget in a xaml tree

<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"

xaml:Class="Napa.Gui.MNWS.MN\_PLTWIN" Name="PltWin">

<Button\_Ok xmlns="clr-namespace:Napa.Gui.MNWS.MN\_STATUSBAR\_Parts"

Name="ProgressBar" m:FormPanel.AttachTop="FORM" ... />

...

</FormPanel>

w = Napa.Gui.MNWS.MN\_PLTWIN.new

#-> nothing printed

This does not call our monkey-patched Button\_Ok initialize method although instance of Button\_Ok is created as part of MN\_PLTWIN construction. This is because Button\_Ok is not constructed by Ruby-code (DLR) but by C# code in WPF-frameworks XAML-loading machinery (CLR).

The fact that sometimes initialize gets automatically executed and sometimes not is artifact of the present imperfect interoperability between DLR and CLR. While understandable, it is inconvenient and can cause confusion and unintended behavior for the GUI client programmer. There are multiple ways this issue can be solved, though none we have found so far are as clean as we would wish. One possibility is to continue using initialize-method for the Ruby-classes but add custom plumbing to the widget construction framework to call it explicitly for the cases where it is not automatically executed. Another possibility is to use own custom initialization method, eg. “constructor”, and call it explicitly for all widgets after CLR-level construction.

This brings us to the end of the technical challenges that have been a part of this transformation project. The next part will deal with the efforts around estimation and scheduling of the project.

8.10 Estimation challenges

Aside from the purely technical problems of making the transformation, there are higher-level problems regarding estimation of effort and schedule for the project.

Developing transformation as a day-to-day work is more akin to debugging than feature-development: One runs the transformation, executes some test case(s) for the transformed code, observes the run-time errors and/or behavior deviating from the original application, digs to find the cause for the error/deviation in the transformed codes, determines how the transformed codes should be changed to fix the error/deviation and modifies the transformation rules to create codes that are modified in the said way. One gets to see further errors only after first fixing the present errors. This makes the planning and tracking of progress of a transformation projection more challenging than for a feature-development project: making an exhaustive list of all fixes still needed to reach the goal of 100% working application is next to impossible. This difficulty naturally extends to project schedule that is very difficult to predict in an exhaustive manner.

If the original codebase is covered with a good set of unit- and integration-tests, one can use those to give estimate of progress as percentage of test pass for the transformed code. Like so many legacy applications, we are not so lucky to have such a test set and developing such at this point has been deemed infeasible because writing tests for GUI-behaviors is very challenging even in general case and even more so for our peculiar combination of NapaBasic + Motif with no out-of-the-box unit-test framework readily available. We have instead resorted to writing unit tests for the transformed GUI directly in IronRuby. While this does not give an equally good picture of completeness, it still enables the lesser goal of preventing regressions when adding new features to the transformation.

Some more crude automatic estimates for completeness can be derived from the transformation stages. We aim to use the following, listed in order of increasing readiness:

1. % of Widgets that are transformed to XAML with no transformation errors
2. % of Widgets that are transformed to valid XAML (compiling)
3. % of Widgets that are transformed XAML that looks like the original widget
4. % of Widgets whose NapaBasic callback code is transformed to Ruby with no transform errors
5. % of Widgets whose that run with no runtime errors
6. % of Widgets whose that look (XAML) and behave (Ruby) like the original widget

Of these 1, 2, 4 and 5 can be fully automated. 3 and 6 require human-based testing and cannot be fully automated, though unit-tests can be written after initial manual testing to ensure the looks and behavior remain correct.

Declan Good writes in his Legacy Transformation article that Transformation is a viable option when the legacy codebase is of reasonably high quality and there is sufficient understanding of it. During our project, both of these requirements have been coming under severe challenge as poorly understood and low quality areas of the codebase have been coming to our attention. This has meant in many cases that refactoring operations to the original codebase have been necessary to proceed with the transformation. These refactoring needs have significantly increased the total work and time needed for the project. However, the positive side is that the higher-quality codebase and greater understanding resulting from the refactorings also directly benefits the present Motif-NapaBasic product in its evolution.

This concludes the challenges we overcame when undertaking this project. After all this hard work it would probably be a good idea to actually look at what the end result of a transformation looks like.

8.11 Example widget

After all these problems and resolutions it would be really nice to see the fruits of our labour. As one example of the about 1700 widget definitions of the Napa Motif GUI, figure 8.2 depicts the LD\_LOAD\_MASS\_DLG dialog, which is one of the widgets related to ship loading conditions design.

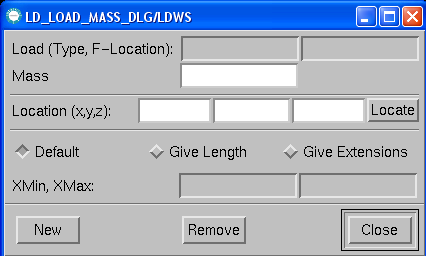


Figure 8.2: The load mass dialog

The Motif-NapaBasic source code of this widget is in file LD\_LOAD\_MASS\_DLG.motif.xml. This is an xml-file that contains Motif widget layouts, properties and associated event-handler NapaBasic code in one rather verbose 800-line package, very much shortened in listing 8.11.

Listing 8.11: The motif widget code

<WidgetDefinition WidgetSet="LDWS" Name="LD\_LOAD\_MASS\_DLG">

<WidgetInstance Class="TEMPLATEDIALOG" Name="LD\_LOAD\_MASS\_DLG" Resource="RES\*LD\_LOAD\_MASS">

<Initialisation>

<Line>@@@ui.close('Help',${sid})</Line>

</Initialisation>

<WidgetAttribute Tech="Motif" Name="AutoUnmanage" Type="Bool">

<Item>False</Item>

</WidgetAttribute>

<WidgetAttribute Tech="Motif" Name="DefaultButtonType" Type="DefaultButtonType">

<Item>DIALOG\_HELP\_BUTTON</Item>

</WidgetAttribute>

<Callback Type="Motif" Name="Popup">

<Line>@{</Line>

<Line>@@Find right load component [LD\_MASS\_PU]</Line>

<Line />

<Line>@if ui.is('MassTable') then</Line>

<Line> @was=arr(3)</Line>

<Line> @was=tp.wareas()</Line>

<Line> @if dm.locate(was,wa) then</Line>

<Line> @currwa=tp.warea()</Line>

...

<Line> @tp.chgwa(currwa)</Line>

<Line> @endif</Line>

<Line>@endif</Line>

<Line>}</Line>

<Line />

</Callback>

<Callback Type="Motif" Name="Ok">

<Line>@{</Line>

<Line>@global</Line>

<Line>@nam='LD\_ADDMASS\_DLG'</Line>

...

<Line>@ui.activate(ui.wid('LdMassOpr',${sid}),'\f' '\'Id\'','load')</Line>

<Line>}</Line>

</Callback>

<WidgetAttribute Tech="Motif" Name="ResizePolicy" Type="ResizePolicy">

<Item>RESIZE\_GROW</Item>

</WidgetAttribute>

...

<WidgetInstance Class="FORM" Name="LMassF">

<WidgetAttribute Tech="Motif" Name="ResizePolicy" Type="ResizePolicy">

<Item>RESIZE\_GROW</Item>

</WidgetAttribute>

<WidgetAttribute Tech="Motif" Name="RightAttachment" Type="FormAttachment" Category="Constraint">

<Item>ATTACH\_FORM</Item>

</WidgetAttribute>

...

</WidgetInstance>

</WidgetInstance>

</WidgetDefinition>

This code is passed through our transformation engine and that results in the XAML code presented in listing 8.12.

Listing 8.12: The resulting XAML

<!--WARNING: This is generated file generated with gui-motif-to-xaml.xslt by RJB.

Any hand-modifications here are likely to be overwritten if/when this file is re-generated -->

<TemplateDialog xmlns="clr-namespace:Napa.Gui.MotifSupport;assembly=CommonGui"

xmlns:xaml="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:m="clr-namespace:Napa.Gui.MotifSupport;assembly=CommonGui"

xaml:Class="Napa.Gui.LDWS.LD\_LOAD\_MASS\_DLG"

m:GUI.Name="LD\_LOAD\_MASS\_DLG"

Title="Load Mass Loads"

DefaultButtonType="DIALOG\_HELP\_BUTTON">

<FormPanel\_LMassF xmlns="clr-namespace:Napa.Gui.LDWS.LD\_LOAD\_MASS\_DLG\_Parts" m:GUI.Name="LMassF">

<FormPanel xmlns="clr-namespace:Napa.Gui.MotifSupport;assembly=CommonGui" m:GUI.Name="WA"

m:FormPanel.AttachRight="FORM" m:FormPanel.AttachLeft="FORM">

<FormPanel m:GUI.Name="Load" m:FormPanel.AttachTop="FORM" m:FormPanel.AttachLeft="FORM">

<Label xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

m:GUI.Name="Label" Content="Load (Type, F-Location):"/>

<InputField\_Text xmlns="clr-namespace:Napa.Gui.LDWS.LD\_LOAD\_MASS\_DLG\_Parts" m:GUI.Name="Text"

IsEnabled="False" MinWidth="72"

m:FormPanel.AttachRight="FORM" m:FormPanel.AttachLeft="WIDGET:Label"

ReplaceOnError="True" InputPattern="$name"/>

</FormPanel>

<Separator xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

m:GUI.Name="S2" m:FormPanel.AttachLeft="FORM" m:FormPanel.AttachRight="FORM"

m:FormPanel.AttachTop="WIDGET:Mass:4"/>

...

</FormPanel>

</FormPanel\_LMassF>

</TemplateDialog>

The next bit of code, listing 8.13, shows the corresponding ruby file that gets generated by the transformation.

Listing 8.13: The generated ruby file

# File generated by TransformNB.m, do not modify by hand

require("gui.rb") # Our ruby GUI framework

require("LoadingConditionsGui.dll") # Assembly with widget XAML

require("wpf/framework\_element.rb") # Extensions on wpf widgets

class Napa::Gui::LDWS::LD\_LOAD\_MASS\_DLG

def loaded\_handler(\*args)

# Find right load component [LD\_MASS\_PU]

# NB: @if ui.is('MassTable') then

if UI.is("MassTable")

was = []

was = TP.wareas

wa = ""

wa = Wpf::Application.current.windows.mass\_table.workarea

if DM.locate(was, wa)

currwa = TP.warea

TP.chgwa(wa)

….

TP.chgwa(currwa)

end

end

end

def help\_handler(\*args)

...

end

...

end

# create classes also for sub-widgets that have code

# so that ruby-methods can be added

module Napa::Gui::LDWS::LD\_LOAD\_MASS\_DLG\_Parts

class FormPanel\_LMassF

...

end

...

end

if ( $0 == \_\_FILE\_\_ ) # If this ruby-file is run directly, show the widget

show\_widget(Napa::Gui::LDWS::LD\_LOAD\_MASS\_DLG) # show\_widget defined in gui.rb

end

Now that we’ve seen our transformation in action I want to share some final thoughts with you before wrapping this up.

8.12 Final thoughts

Evolutionary iterative development is considered a best practice today in order to minimize risk and maximize value of developed features for a large codebase. But when there is a need to move away from some legacy languages and libraries, language barriers can form a difficult gap to cross in such an iterative way. Migration from some .NET language (say VB to C#) can still be done by migrating file-by-file, all the time having a working product. But when the source and destination languages, libraries and GUI-toolkits are two decades apart, bit-by-bit migration becomes close to impossible. The remaining alternatives are a full rewrite or legacy transformation. Neither is an easy choice and high risks are present on both paths, though the risks are of a different nature: mainly risks of extreme technical challenges in the transformation and risks of extreme programmer resources needs in a rewrite. A large codebase and a continuous need to add features to the legacy product strongly favor the transformation alternative if its technical challenges can be overcome.

In our 1.5 year long legacy transformation path we have faced these challenges, many of which no apparent solution has been seen during time of discovery. But we have learned a lot and some resolution to all of the problems has been found so far. This has increased our confidence that solutions will be found also to the forthcoming yet-to-be-discovered problems that surely still lie ahead. Hence we can say that one should not too hastily conclude a legacy transformation project “impossible” even when there are problems that lack a solution for an extended time.

Meanwhile we have learned to use several strategies to overcome problems that are not feasible to solve by the default way of writing new transformation rules. The dynamic flexibility of IronRuby has already shown to be an important enabling factor here, saving the day several times with metaprogramming possibilities that have allowed us to patch the gap between conflicts raising from the combination of WPF, XAML and Napa Motif GUI. And I expect to see more instances of “Rubyish” features coming to save us from blocking before we hit the finish line.

To further ease the transformation and increase the quality of the WPF-Ruby GUI, we have selected a small subset of commonly used low-level widgets (such as file open/save and printing dialogs) to be rewritten directly in XAML+Ruby instead of being included in the transformation. The widgets to be rewritten must be relative stable in functionality to avoid the problem of “shooting a moving target” when making a new duplicate implementation of them. Good rewrite candidates also form a very small part of the codebase, but still are encountered by the user a lot, so redesigning them to fully use WPF-features can have a relatively bigger positive effect on the user experience.

When the automated transformation and its required framework is complete and the resulting XAML/Ruby -files adopted as first-class source files, we have still lots of work to do. Napa GUI renewal will then continue along two directions to tap the full potential of the new platforms. First is the evolutionary development of the transformed widgets and their logic. This should include, in addition of adding new features, further refactoring to take advantages of the new technology. One such technology is Object-Orientation: NapaBasic does not support custom classes and objects and hence the GUI-logic is rather procedural in nature and prone to duplication. There is no magic wand we can wave in the automatic transformation to make procedural code object-oriented: human programmers need to apply their higher-level thinking in individual cases to do that. The second approach is development of new widgets (and replacements of old widgets) from scratch and their integration to the existing set of transformed and hand-coded widgets. In this way we can re-write parts of GUI step-wise with low risk and continuous releases. In both of these directions, we expect the powerful features and wide support of WPF- and IronRuby-technologies to raise the productivity of Napa SW development to new levels.

Robert Brotherus - Software Architect - [www.napa.fi](http://www.napa.fi)

8.13 Summary

I would like to thank Robert for his contribution, it has been most valuable to learn about their transformation project for me. This project really shows how many systems are being built these days, with a mix of languages and technologies. It also shows that there are options beyond the “Big Rewrite”.

One of the benefits of using a language like IronRuby, which is supported by the DLR, is that you have an AST readily available without a need to implement one yourself. Then using a dynamic language really helped them overcome some of the challenges presented by converting a project that is developed in technologies that can be truly called legacy to a contemporary toolset like WPF/Ruby/.NET is quite impressive.

Our next chapter will deal with extending IronRuby from C# with types that act as if they are native ruby types. This will show you how the libraries for IronRuby have been developed.