A USER INTERFACE MANAGEMENT FRAMEWORK TO ASSIST IN PRODUCT DEVELOPMENT HARDWARE-IO USER-INTERFACES

Ad van Gerven

Alten Nederland BV

Abstract

When designing effective user-interfaces for smart products, there is a need to minimize the coupling between the functional software application and the physical user-interface of the product. The Bridge software design pattern (Gamma, Helm, et al., Structural Patterns, 1995) provides an obvious means to reach this goal. This article presents a framework that encapsulates base classes for the application's functional software and for the product's user interface-implementation, together with the Bridge pattern methodology. A user-interface manager package is added to the framework to support attaching (and detaching from), of different user-interface implementors to the product's functional software, dynamically and at run-time.

A USER INTERFACE MANAGEMENT FRAMEWORK TO ASSIST IN PRODUCT DEVELOPMENT HARDWARE-IO USER-INTERFACES

In the design and development of our MIDI devices and controllers we tend to start with prototyping in an early phase, while specifying and implementing the actual physical user interface in a late phase. The current way of working can be maintained and its impact on various parts of the project can be minimized by decoupling the actual user interface as much as possible:

- This will also allow to design and try-out various user interface alternatives that can be compared in an early phase;
- During development, and especially when designing the actual physical user interface, we want to be able to connect to multiple different user interfaces simultaneously, e.g. both the software and the physical user interface;
- In addition we would like alternative user interface implementors, for local auto testing and for remote diagnostics at a customer location;
- Finally, not as a primary goal but as an expected bonus, this will make porting of the product's functionality to different platforms (standalone hardware, desktop or laptop, iOs and Arduino) easier.

The considerations above have led to the development of a User Interface Management Framework, which allows us to actually use all benefits listed above, and which is described in this article. As we discovered this framework is also very useful, with some minor modifications, for abstraction of the functional application software from other system interfaces (like MIDI and other Machine-Machine interfaces) as well.

Application view on the framework

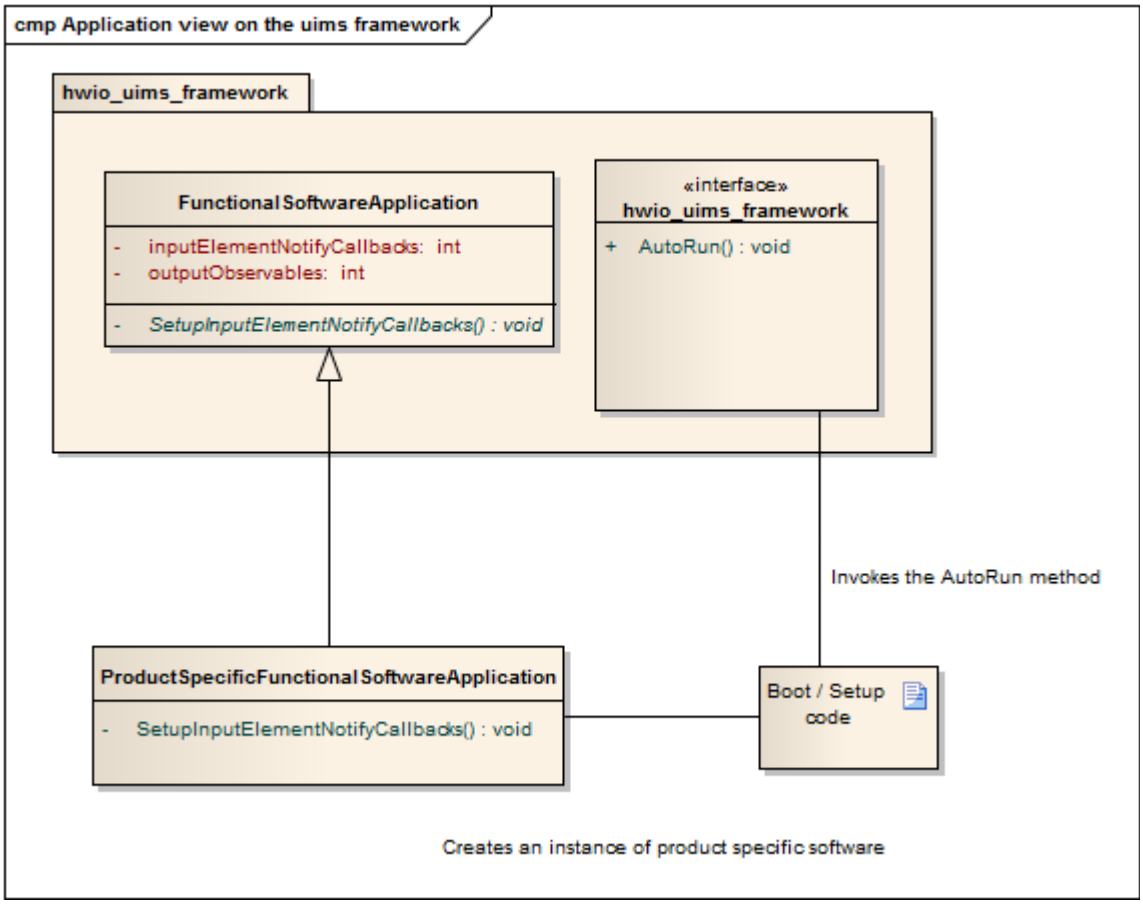

Figure 1 Application view on the hardware-I/O User Interface Management Framework

As shown in the diagram above the User Interface Management Framework (UIMF) is modeled as a package with a simple interface, which consists of a base class: FunctionalSoftwareApplication and a method: AutoRun.

The client or application using the framework must provide a short section of code, which does the following:

- Define a subclass of the FunctionalSoftwareApplication base class;
- Create an instance of the subclass;
- Invoke the AutoRun method of the UIMF while passing:
    - The instance of the subclass;
    - The file name of a hardware I/O Bill Of Materials file;

    Optionally a list of user interface implementor module names may be passed to the AutoRun method.

An example is given in (Example A of a simple client application of the UIMF, p. 12).

The Bill of Materials

A list of all user-interface elements must be available for each specific product. This list is used / shared by the functional application software and the UIMF and, possibly, with other production tools such as a layout designer, front panel milling tool etc. Each element on the list must be uniquely identifiable. For each element on the list a type (such as lamp, button, etc.) must be specified. This list is a subset of the product's full component list or Bill of Materials (BOM). This subset will be referred to in this document as hardware-I/O BOM.

The hardware-I/O BOM is essentially a comma separated value (.csv) file. The first row specifies all column headers or record field names. (Table 1, p. 5) lists common fields, which are available to all clients of the BOM.

Table 1 Common fields in a hardware I/O BOM

Field name	Value	Remarks
elementIdentifier	A unique, non-empty value is mandatory	Uniquely identifies an individual user interface element
elementTypeIdentifier	A non-empty value is mandatory	Uniquely identifies the type of an individual user interface element
elementLabel	A non-empty value is optional	When specified, the text is displayed on top of the individual element in a software user interface. In a physical user interface, when specified, the text is printed on or engraved in the individual element

The hardware I/O BOM has one reserved element type identifier, named 'INFO'. User interface elements of this type have neither input nor output parameters and are therefore ignored by the functional software application. These elements can be used to display additional non-interactive text and / or graphics on the user interface.

In addition to the common fields defined in (Table 1, p. 5), the hardware I/O BOM may contain user interface implementor specific fields. In the examples (Example A of a simple client application of the UIMF, p. 12) and (Example B of a simple client application of the UIMF, p.

14) these specify grid layout manager properties (e.g. row, column, rowspan and columnspan) for a tkinter user interface.

The user-interface elements

The framework supports compound user-interface elements:

- Each element can have any number (zero or more) of input parameters, each of which is uniquely identified;
  The input information flow is initiated by the operator (by entering data, clicking buttons etc.) and is passed on to the functional application software.
- Each element can have any number (zero or more) of output parameters, each of which is uniquely identified;
  The output information flow is initiated by the functional application software and is passed on to the user interface and its elements.
- Each element parameter can be an input parameter, an output parameter or both;

Design view of the framework

The internals of the framework correspond largely to the Bridge design pattern, which is meant to decouple an abstraction from its implementation so that the two can vary independently (Gamma, Helm, et al., 1995 Structural patterns). In our case the abstraction is the software interface used to connect the application functional software to the actual hardware user-interface of a smart product.

This abstract software interface is named UiAbstractorInterface in Figure 2. The FunctionalSoftwareApplication base class connects to this interface, and the base class for the actual hardware user-interface, named UiImplementorBase in Figure 2, implements the UiAbstractorInterface. This allows the functional software to operate with many different implementations, such as a Tkinter GUI implementation or an RPi.GPIO implementation of the products user-interface without having to know the details of that implementation.

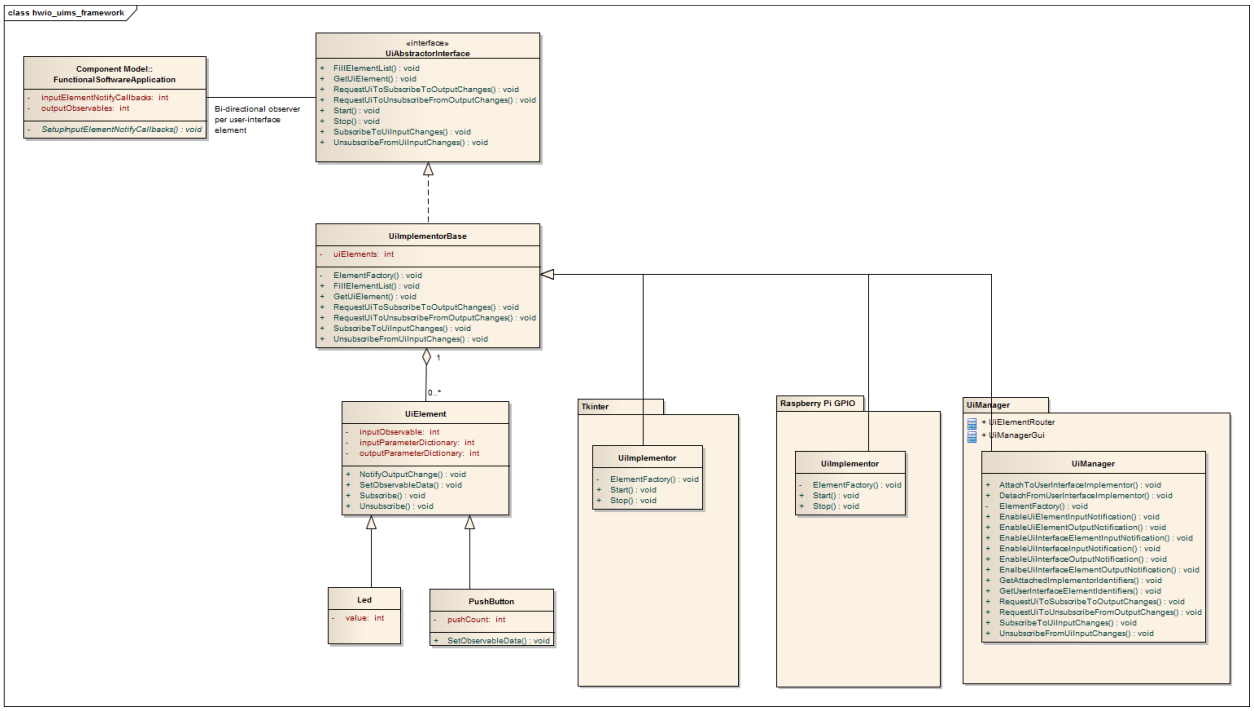

Figure 2 Class diagram for the hardware-I/O UIMF

It must be noted that any software library or application can implement this abstract interface, even when the implementor does not contribute to a user interface at all. Of specific interest are of course auto testers and record and play-back facilities.

The user interface manager

The framework provides one such special implementation of UiImplementorBase named UiManager. The UiManager package provides much of the framework's added value, since it allows dynamic run-time attaching (and detaching) of different user-interface implementors to the functional software. The functional software never knows how many or which user-interface implementors it is connected to. The package also has a UiManagerGui application, which provides a run-time graphical user interface for attaching and detaching of actual user-interface implementors (Figure 10, p. 15).

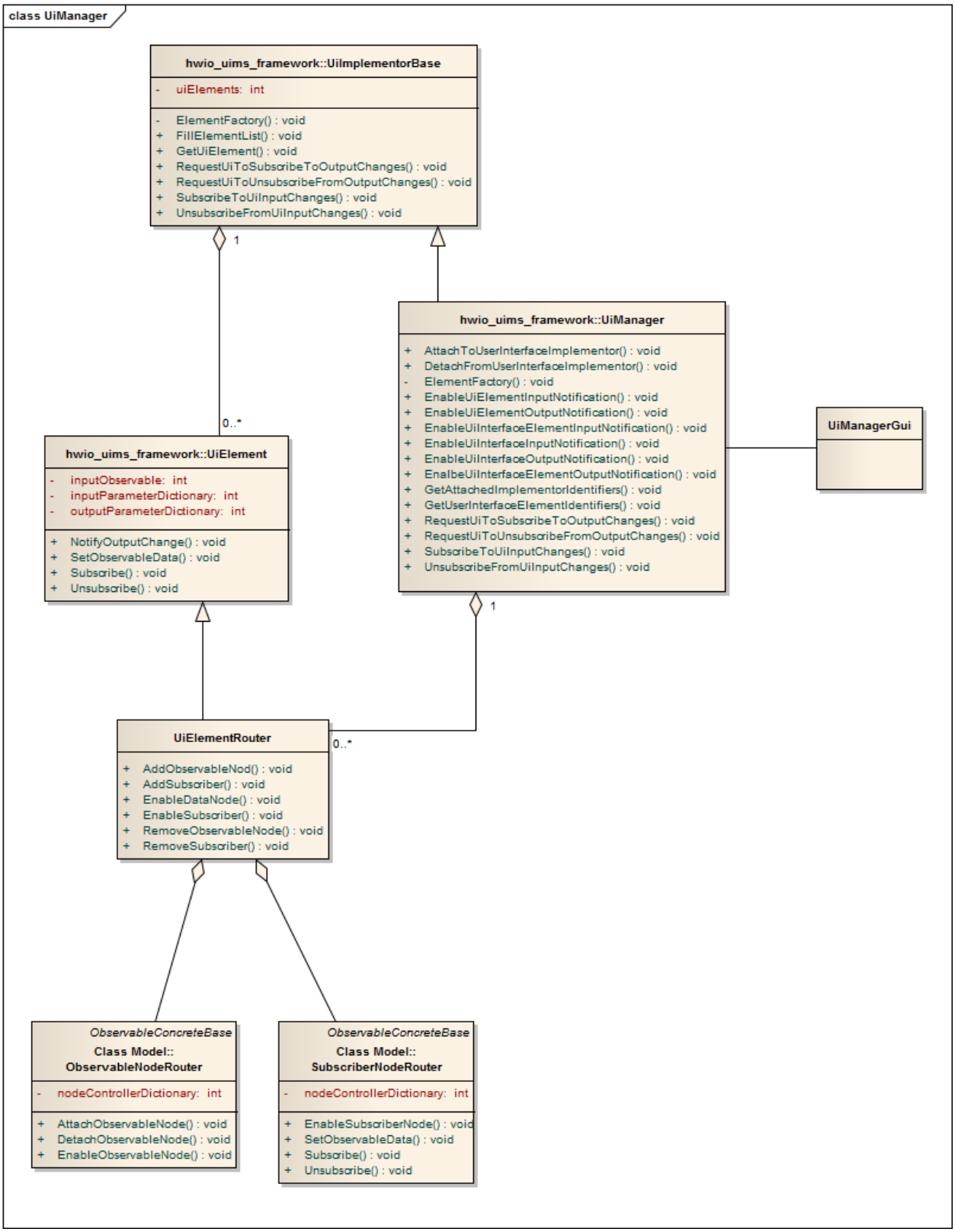

Figure 3 The UiManager package (class) diagram

The UiMerger class has, instead of actual user-interface elements, a special user-interface element named UiElementRouter for each elementIdentifier named in the hardware-IO Bill of Materials. Instances of the UiElementRouter class are fully unaware of the actual user-interface element's responsibilities. Instead they serve as N-to-M gateways connecting the functional software to any number of actual user-interfaces.

Each instance of UiElementRouter has an aggregate instance of the ObservableNodeRouter and an aggregate instance of the SubscriberNodeRouter class. These are specializations of the of the ObservableConcreteBase class.

The ObservableControlledMerger class allows 1 observer instance, in this case an instance of FunctionalSoftwareApplication, to subscribe to multiple observables, in this case real user-interface implementors, for changes in user-interface element input data. The class provides methods to enable or disable notification for each of the observable nodes individually and at run-time.

The ObservableControlledDispatcher class allows multiple observer instances, in this case real user-interface implementors, to subscribe to a single observable, in this case an instance of FunctionalSoftwareApplication, for changes in user-interface output data. The class provides methods to enable or disable notification of each of the subscribers individually and at run-time.

References

Gamma, Helm, Johnson & Vlissides (1995). *Design Patterns*. Addison-Wesley, ISBN 0201633612

Abbreviations

Abbreviation	Description
BOM	Bill Of Materials
GPIO	General Purpose Input / Output
GUI	Graphical User Interface
I/O	Input / Output
LED	Light Emitting Diode
UI	User Interface
UIMF	User Interface Management Framework

Example A of a simple client application of the UIMF

```
from user_interface_management_framework import
FunctionalSoftwareApplication, AutoRun

class UimfExample(FunctionalSoftwareApplication):
    '''The actual functional software application.
    '''

#--------------------------------------------------------------------------
    def __init__(self):
        # Initialize the own instance of the base class
        FunctionalSoftwareApplication.__init__(self)

        # Specific properties
        self._ledIsOn = False


#--------------------------------------------------------------------------
    def _SetupInputElementNotifyCallbacks(self):
        '''Specific setup of input parameter notification callbacks.
        '''
        self._inputElementNotifyCallbacks['BUTTON_01'] =
            self.NotifyChanges_Button_01


#--------------------------------------------------------------------------
    def NotifyChanges_Button_01(self, notifyCallback):
        self._ledIsOn = not(self._ledIsOn)
        self._outputObservables['LED_01'].SetObservableDataNotifyChange(
            {'value' : self._ledIsOn})

if __name__ == "__main__":
    fsa = UimfExample()
    AutoRun(fsa, 'uimf_example.csv', ['tkinter_implementor'])
```

Figure 4 Example of a UIMF application, implemented in Python

This application has a very simple user interface, which consists of 1 input element ‘BUTTON_01’ and one output element ‘LED_01’. These elements are defined in the hardware-I/O BOM file, named ‘uimf_example.csv’, which is shown below:

```
elementIdentifier,elementTypeIdentifier,elementLabel,grid row,grid
column,grid rowspan,grid columnspan
BUTTON_01,PushButton,Power,0,0,1,1
LED_01,BinaryLed,,0,1,1,1
```

Figure 5 The hardware I/O Bill Of Materials used in this example

The information in the hardware I/O BOM is shared between the functional application software and the UIMF.

The application code in (Figure 4, p. 12) defines a subclass, named UimfExample of the UIMF base class FunctionalSoftwareApplication. It defines an implementation for the base class

method `_SetupInputElementNotifyCallbacks`, in which a notification callback is specified for the BUTTON_01 user interface element. This notification callback is invoked by the framework whenever the associated button is clicked. The notification callback toggles the state of the LED_01 user interface element.

Subscription of the UimfExample instance to the user interface's input elements is implicitly performed by the UIMF when the AutoRun method is invoked. Subscription of the user interface's output elements to output parameter changes by the FunctionalSoftwareApplication is also implicitly performed by the UIMF when the AutoRun method is invoked.

When invoking the AutoRun method, the application code (Figure 4, p. 12) specifies an array with the name ('tkinter_implementor') of one user interface implementor module. This specific implementor module is part of the framework and has implementations for all user interface elements in the hardware I/O BOM (Figure 5, p.12).

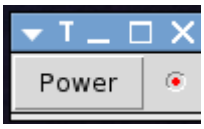

Figure 6 Active user interface for the UIMF example application

Example B of a simple client application of the UIMF

This section describes a somewhat more extended example. The user interface has 8 elements: 6 LEDs (output elements) and 2 push buttons (input elements):

Figure 7 Example of a product specific BOM

```
elementIdentifier,elementTypeIdentifier,grid row,grid column,grid rowspan,grid
columnspan,g2 row,g2 column,g2 rowspan,g2 columnspan,g3 row,g3 column,g3 rowspan,g3
columnspan
LED_01,BinaryLed,0,0,1,1,0,0,1,1,0,0,1,1
LED_02,BinaryLed,0,1,1,1,1,0,1,1,1,1,1,1
LED_03,BinaryLed,0,2,1,1,2,0,1,1,2,2,1,1
LED_04,BinaryLed,0,3,1,1,3,0,1,1,3,3,1,1
LED_05,BinaryLed,0,4,1,1,4,0,1,1,4,4,1,1
LED_06,BinaryLed,0,5,1,1,5,0,1,1,5,5,1,1
PUSH_01,PushButton,1,0,1,3,0,1,3,1,6,0,1,3
PUSH_02,PushButton,1,3,1,3,3,1,3,1,6,3,1,3
```

Please note that this hardware-I/O BOM has more columns than that of (Figure 5, p. 12). The BOM in (Figure 7, p. 14) specifies layout positions for 3 independent and different tkinter implementors.

The file in Figure 7 contains 8 elements: 6 LED's and 2 switches. In this example all LED's are of type BinaryLed and all switches are of type PushButton. In this example the "elementTypeIdentifier" values are simple, easy to understand terms like BinaryLed and PushButton. In a more practical situation the element type identifiers would be catalog numbers for the actual hardware elements (if any) that are to be used, such as "L 53 LGD" for BinaryLed:

Figure 8 Example of a red 5mm led of type L 53 LGD

The remaining fields in the UiElementsBom file, e.g. the numbers in each row in Figure 7, specify the properties for a grid layout manager per element for each of the user-interface implementors.

In this particular case 3 user-interface implementors can be applied. One of the implementors has a diagonal layout of the LEDs, another has a horizontal layout and the last one has a vertical layout of the LEDs, as shown below.

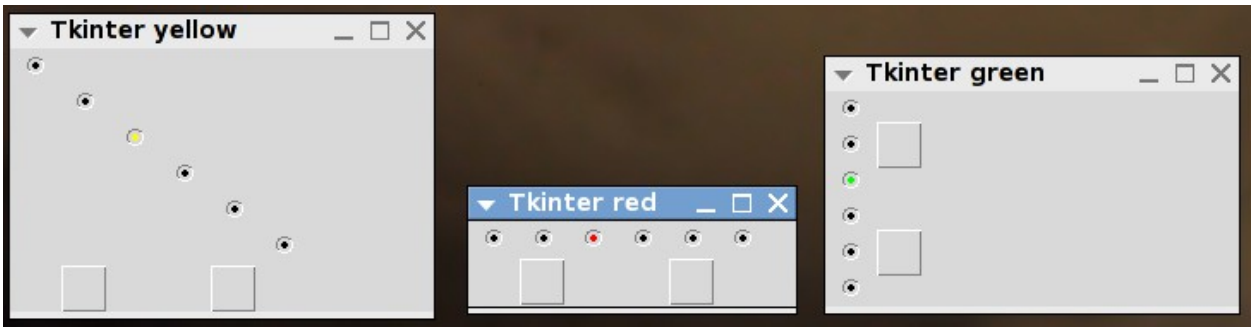

Figure 9 Example of 3 user interface implementors sharing a single BOM

Each of these user interfaces can be attached or detached, dynamically and at run-time, using the hardware-I/O manager provided by the UIMF.

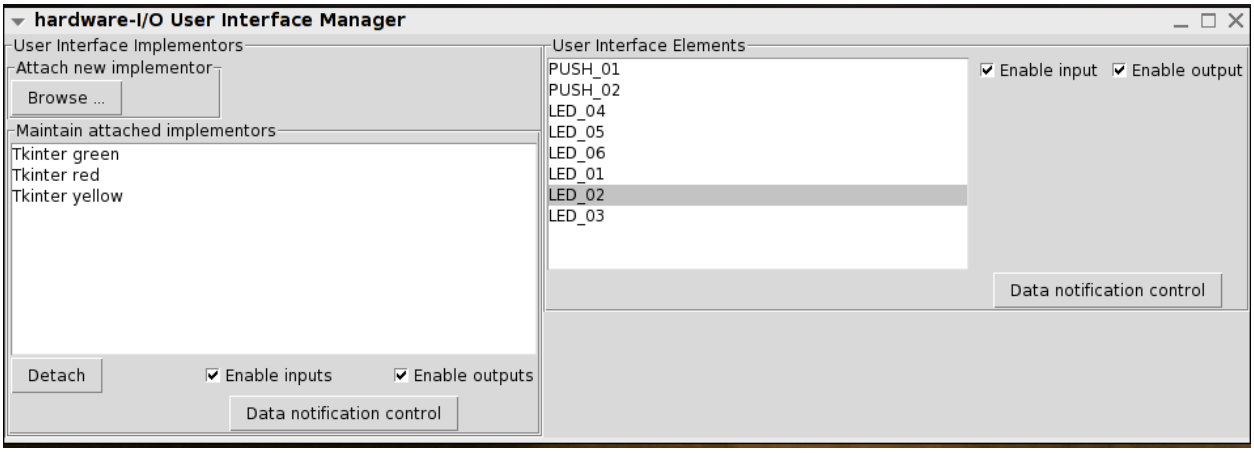

Figure 10 A snapshot of the UIMF hardware-I/O manager

In this particular example the FunctionalSoftwareApplication is associated with the hardware I/O user interface manager, and is unaware of the actual physical user interfaces that are active at any one time.