

Interfaces in Rhapsody® in C

Techletter Nr. 1

WILLERT.

This is the first Techletter of Willert Software Tools.

What is a Techletter? Well actually nothing more than a newsletter, just that the content mainly focusses on technical items in the UML and in embedded systems with limited resources.

Interfaces in Rhapsody® in C

Why Interfaces ?

Some years ago it was normal that a single developer in a software department programmed all the software for that department. Nowadays this would be an exceptional situation, even smaller projects consist of multiple developers. This way of working demands, apart from the existing challenges that we are coping with already, one more challenge: Cooperation.

A major part of the extra manpower is used by cooperating while a lot of communication is necessary. The project must be tuned regularly or otherwise developers are running in opposite directions.

Unfortunately the management of many companies have no real understanding for this. They often are convinced that a one man-year project can be done in 1 hour by 1920 developers in Bangalore.

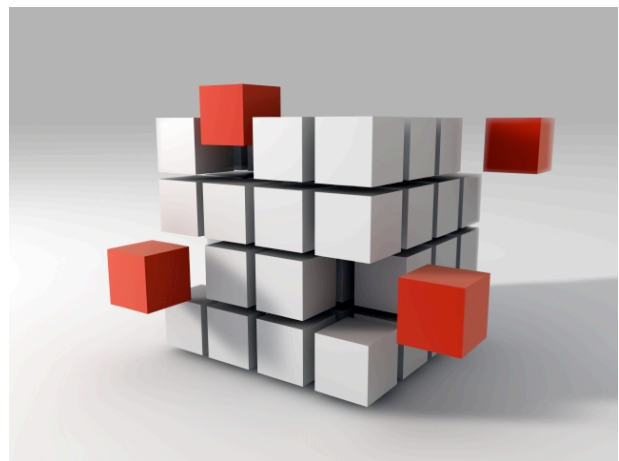


The secret lies in dividing the project in smaller, mainly independent components. And off course in the earliest possible definition of the interfaces between these components. I use the word component here although the UML also knows Components. This is confusing since they are not equal. I will use the term "software components" to avoid confusion.

As soon as the interface between 2 software components is determined and frozen, developers can work

independently from each other. Development is a lot safer and unwanted side-effects can be avoided.

It was Tom DeMarco who already said: The most important thing in a project is the definition of interfaces. Without this definition every project is doomed to fail. (BTW: The man-year example is also Tom DeMarco's).

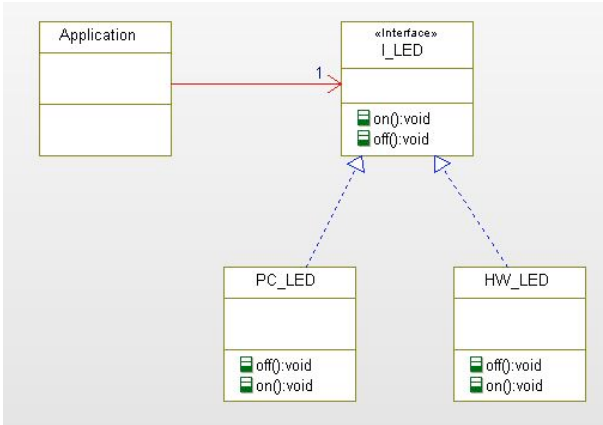


A further argument for the division of the project in software components lies in the steady increasing pressure to deliver projects faster and faster. (due to a disease called: Time to Market) This causes another decrease of the already limited time for the projects and developers are forced to build their software in such a way that at least parts can be re-used.

The importance of a correct and thorough interface description should be clear here, or not :-)

What is a correct interface description? In software this should contain a description of all function calls with parameters and return values. Also included is a description what the function actually does, or better, what it's responsibility is. A function call is a form of synchronous communication. Next to synchronous communication, the UML also knows asynchronous communication. This consists of a description of events and, if necessary, their parameters, that a software component can receive. Sounds plausible but is mostly totally different from function calls.

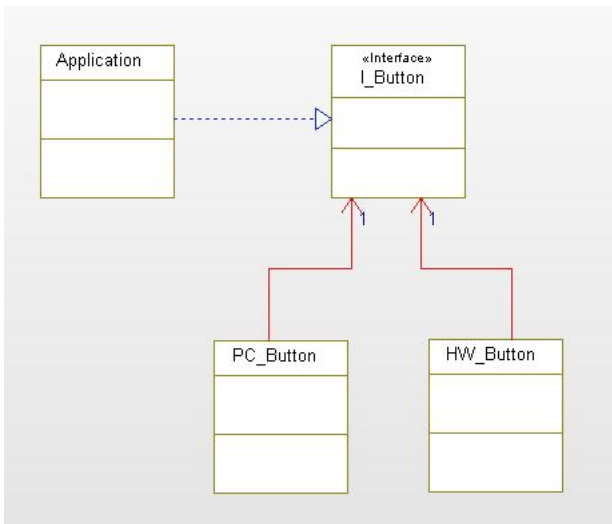
With Inheritance



To clarify the last point we will look at the above diagram. There we see the UML definition of the interface of an LED. This interface is used by the class "Application".

A simple representation of a real-life project. The class Application now knows, because it knows the interface I_LED, what an LED can do exactly. There are 2 public functions, on() and off(), that can be called by other classes (hence public)

For the I_LED interface there are 2 possible implementations, one for a PC and one for hardware.



The application couldn't care less which of the implementations it really calls, it only calls the functions defined in the interface definition. Actually this could easily be implemented with using inheritance, later we will look at a small trick how this can be done quite elegantly.

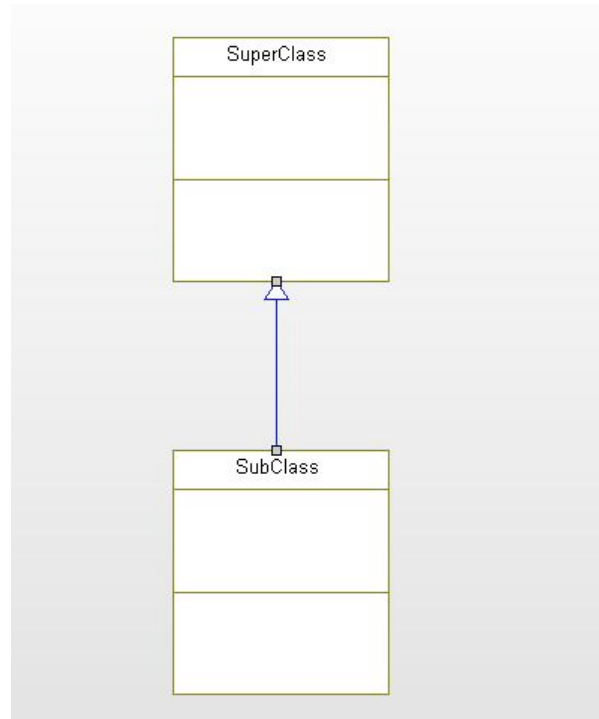
The LED class should be a re-usable software component. For this type of components a beautiful solution. However, this changes when the software component contains active elements, that means the software component sends events when certain situations occur.

The problem there is that the events to send and the recipient of these events must be known to the sender. That means that the relationship between the class and the interface must be made in the other direction and for

that inheritance is used to implement that correctly (See previous picture)

Not so very long ago this was not possible in Rhapsody in 'C', just in C++ and Java. Luckily Rhapsody has, since version 7.0, the possibility to generate code for the inheritance of interfaces. A nice and elegant method to describe interfaces but there are other methods that sometimes offer additional advantages.

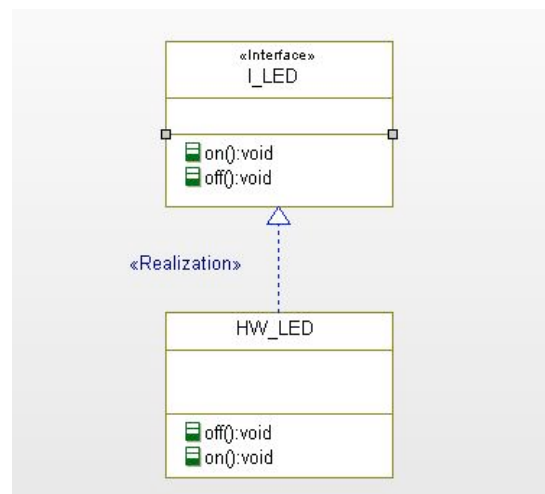
Rhapsody in C does not offer complete inheritance, only interface inheritance is implemented.



Inheritance of classes can be drawn graphically, however, when generating code a warning appears that the inheritance will be ignored. Not as tragic as it looks, interface inheritance already gives us some great benefits.

How does it work?

To figure that out we will implement a small example and take a look at the generated code.



I_LED defines that an implemented LED has 2 functions, on() and off(). In Rhapsody in C public operations have the class name as a prefix, otherwise the C compiler would not be able to resolve the multiple defined names. The function calls look as follows:

```
I_LED_on(me->itsLED);
I_LED_off(me->itsLED);
```

The generated code will implement this with the help of a function pointer table, that connect this call with the correct ,C' function. The declaration of the class looks as follows:

```
/*## class I_LED */
typedef struct I_LED I_LED;
struct I_LED {
    const I_LED_Vtbl * I_LEDVtbl;
};
```

The Vtbl is filled with the correct values in the constructor of the class (Called Init in Rhapsody in C).

```
/*## class I_LED */
void I_LED_Init(I_LED* const me, const
I_LED_Vtbl * vtbl) {
    me->I_LEDVtbl = vtbl;
}
```

The call of a function in the interface is implemented in such a way that the correct function to be called is determined and a correct "me" pointer is calculated. That is necessary because classes have different structures depending on what they do exactly (active, state-chart. Monitor) So the me pointer does not always point to the same place.

```
/*## operation off() */
void I_LED_off(void * const void_me) {

    I_LED * const me = (I_LED *)void_me;

    if (me != NULL)
    {
        I_LED_Vtbl* vtbl = (I_LED_Vtbl*)
            (me->I_LEDVtbl);

        if ((vtbl != NULL) &&
            (vtbl->I_LED_off != NULL))
        {
            size_t addr = (size_t)me;
            void* realMe =
                (void*)(addr - vtbl->I_LED_offset);
            (*vtbl->I_LED_off) (realMe);
        }
    }
}
```

Let's take a closer look at the implementation of HW_LED. The structure contains an instance of the I_LED class.

```
/*## class HW_LED */
typedef struct HW_LED HW_LED;
struct HW_LED {
    /*##[ ignore */
    struct I_LED _I_LED;
```

```
/*##]*/
};

The structure is filled in the Init function,

/*## class HW_LED */
void HW_LED_Init(HW_LED* const me) {
    /* Virtual tables Initialization */
    static const I_LED_Vtbl
        HW_LED_I_LED_Vtbl_Values = {
        offsetof(HW_LED, _I_LED),
        (void (*)(void * const void_me))
            HW_LED_off,
        (void (*)(void * const void_me))
            HW_LED_on
    };
    I_LED_Init(&me->_I_LED,
        &HW_LED_I_LED_Vtbl_Values);
}
```

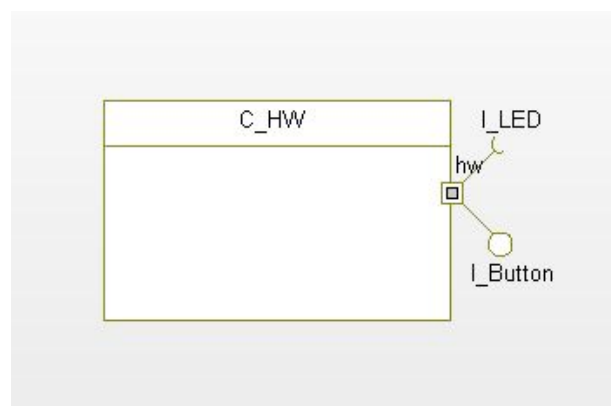
The declaration of the Vtbl is (static) placed in the constructor; all necessary pointers and offsets are determined here directly.

The real content of the off() and on() functions is placed in the HW_LED_on() and HW_LED_off() body.

Conclusion: the code is a little bit bigger; it takes slightly longer to execute but in exchange the construction is very readable and understandable.

Ports, another possibility.

Another method to implement interfaces is ports. Since the UML 2.0 ports are a part of the UML. If a class communicates it can have a port that is connected with interfaces. There are 2 kinds of interfaces, required and provided, just like in our previous example with interfaces. Rhapsody in C also support ports. Because part of the port implementation is solved in the framework, it must be implemented in there. All Willert frameworks support ports.



Ports are in fact just a simpler and more elegant way of describing classes with interfaces. There are, however, differences. For instance: Ports are connected in run-time. This method will use time and memory and is also relatively complex to implement. With interfaces the user must take care of the connection between the communicating objects. Ports will do this automatically. An event or a function call are not performed in another object, they take place in the port itself.

```
RiCGEN_PORT (me->hw, evPressed() );
```

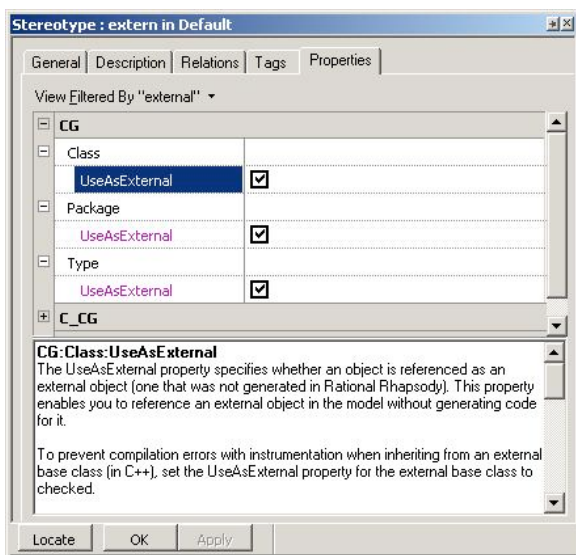
In the framework there is code that takes care that the event will land in the correct place.

This causes that ports are less suitable for hard real-time applications and also not for situation where a continuous battle is fought for the last byte. On top of that the current implementation of ports makes extensive use of recursion, what could be problematic for a certification according to SIL or DAL. The representation however is very understandable.

Extern, simple and good

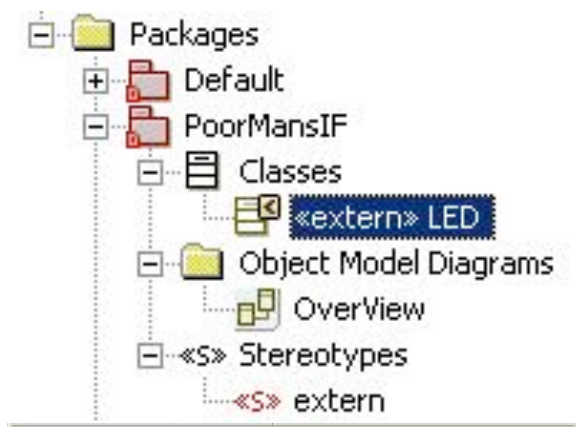
Do we really need ports and/or inheritance? The answer is short and clear: No. It is possible in an easier way, perhaps graphically not as elegant but with less code overhead and less complexity.

This works with a small trick. Rhapsody can apply the property "UseAsExternal" to the elements class, object and type. That means that this element is there but that Rhapsody should not generate code for it. To make this more visible we should apply this property to a stereotype.

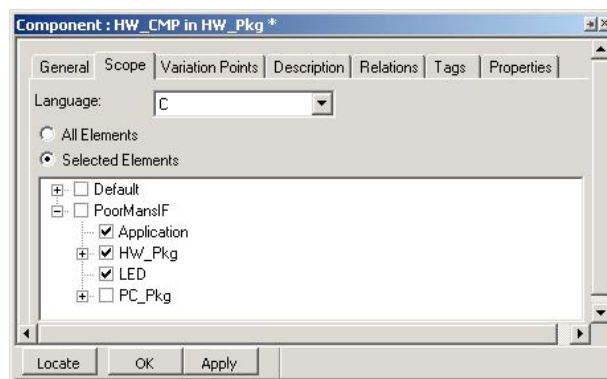


If this stereotype is applied to an element, a small graphical element appears in the browser to indicate that this is an external element.

As already mentioned, Rhapsody will not generate code for this element, it just assumes that it is there and leaves

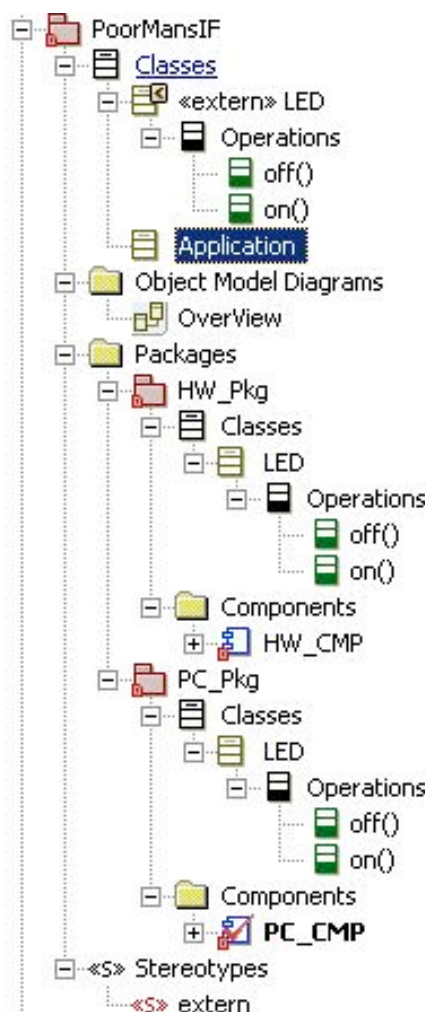


it to the user to take care of that. Here our small trick kicks in: We will take care that in our model, in another place (package) for every implementation another element LED is there. With the help of the scope in the component we can tell every component exactly which LED it should implement.



We need additional packages because the same element (or an similar element with the same name) can not reside in the same package.

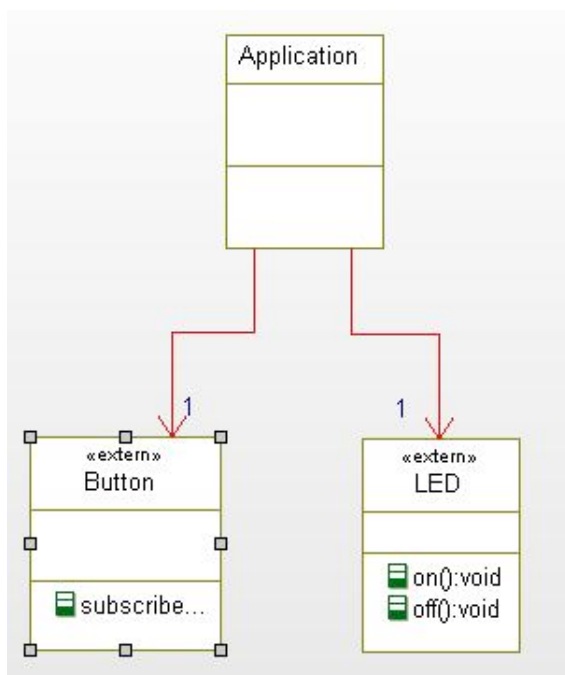
A disadvantage is that these elements all must have the same name otherwise it won't work. Also the option that Rhapsody generates a separate directory for every package can not be used. Otherwise LED.h can not be found.



Here is an overview how this solution looks in the Rhapsody browser. The application can have a relation with our external LED class and call its functions. After generating and compiling the correct function is called automatically.

Publisher Subscriber

The previous example works like a charm and does not generate a single byte of extra code. Unfortunately it is not directly applicable when events must be send. Then we have the same problem as we already had with the interfaces. Fixed names of classes are know that must be used as software component With interfaces we were able to reverse the direction of the relation. here we can not do just that. We can not, just like that, create an external application class. But we can use a mechanism known in C as call-back function. It actually is a design-pattern that is also known as publisher-subscriber:



A class that can send events offers a subscribe function where every class (In this simple example only one) can register itself. This is done by passing a pointer to a private function that sends an event to the class itself.

```
Button_subscribe
    (me->itsButton, sendEvPressed, me);
```

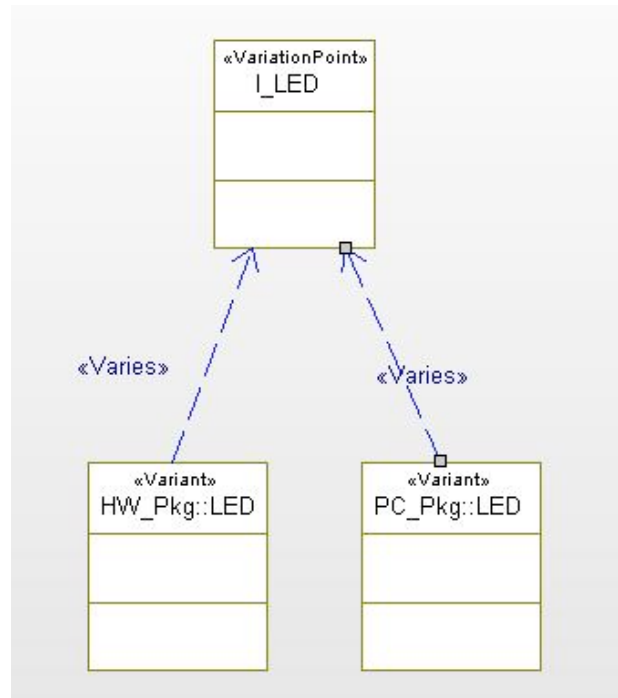
Additional the caller passes a pointer to itself (a "me" pointer). We need that for calling the call-back function. The publisher class then stores the parameter and will take care that this function is called as soon as the events occurs.

```
if ( me->PS_cb != NULL )
{
    me->PS_cb ( me->PS_me );
}
```

In this way the application knows the hardware component that, in its turn needs no knowledge about the application.

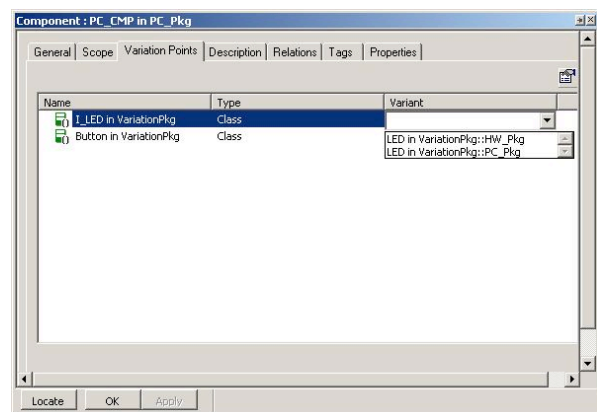
Variation Points

Since Rhapsody 7.5 the previous "trick" is being formalized and available as feature. It is called "variation points" and can be implemented with the help of provided stereotypes.



The interfaces are defined as variation points with the help of stereotypes. All implementations have a dependency with this variation point. The dependency contains the stereotype "varies". Also the implementations have the stereotype "variant".

That will tell Rhapsody that this is an implementation of the interface. In the respective component is chosen which of the variations should be generated.



Rhapsody generates a special include file I_LED.h that contains macro's for all function calls that point to the right function. The difference with the external method is that the names MUST be different otherwise it won't compile. Generating packages in directories also works.

Implementing the publisher-subscribe design pattern is also possible in the same way as in the external example.

Conclusion

What method should be used then. As usual the answer depends on a lot of facts. If there is no certification (61508, DO178b or other) that must be done, you can just take whatever you like. Ports and interfaces are very elegant and greatly increase the understandability of the model.



If a certification process must be done or if the code must be MISRA compliant or when memory and or run-time are bottlenecks, ports and interfaces are not the method of choice. At higher SIL and or DAL levels even variation points are difficult since they use preprocessor macro's. in that case only the poor-man's interfaces are left. But when using them, the model is much better to read than a model were everything is implemented directly.



UMLforum.de

Published by

WILLERT SOFTWARE TOOLS GMBH
Hannoversche Straße 21
31675 Bückeburg
Tel.: 05722 - 9678 60

www.willert.de info@willert.de

Tel.: +49 5722 9678 - 60

Author: Walter van der Heiden