

Interfaces in Rhapsody® in C

Techletter Nr. 1

WILLERT.

Dies ist die erste Techletter der Firma Willert Software Tools.
Was ist eine Techletter? Eigentlich nichts anderes als eine Newsletter, nur dass der Inhalt sich vorwiegend auf technische Dinge auf dem Gebiet der UML und auf kleine Ressourcen-beschränkte embedded Systeme bezieht.

Interfaces in Rhapsody® in C

Warum Interfaces

War es vor vielen Jahren noch normal, dass ein einziger Entwickler in einer Softwareabteilung die ganze Software programmiert hat, so ist es derzeit immer mehr eine Ausnahmesituation. Sogar kleinere Projekte bestehen immer häufiger aus Teams mit mehreren Entwicklern. Diese Arbeitsweise verlangt neben all den bestehenden Herausforderungen, die wir ohnehin schon haben, eine weitere: Zusammenarbeit.

So geht ein großer Teil der Manpower schnell wieder verloren, weil viel Kommunikation notwendig ist. Das Projekt muss häufig abgestimmt werden, sonst gehen die Entwickler in verschiedene Richtungen.

Die Vorgesetzten haben dafür leider wenig Verständnis, sie denken immer noch, dass ein Projekt von einem Mannjahr durch 1920 Entwickler in Bangalore in einer Stunde erledigt ist.

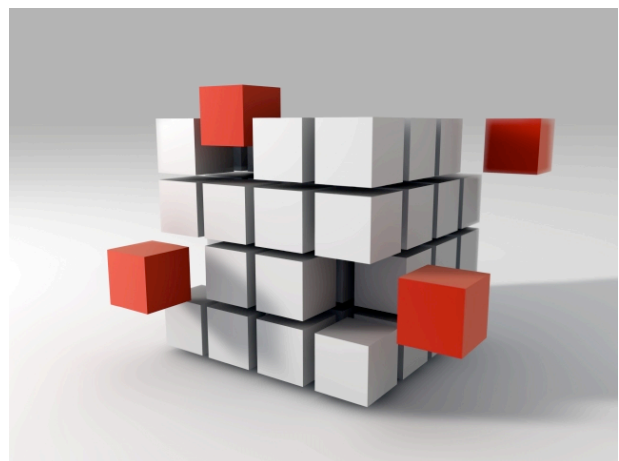


Das Geheimnis liegt hier in der Aufteilung des Projektes in kleinere, größtenteils selbständige Komponenten. Und natürlich in der rechtzeitigen Festlegung der Schnittstellen zwischen diesen Komponenten. Ich verwende hier das Wort

Komponenten, leider kennt die UML auch Komponenten (oder Components) das ist ein wenig verwirrend, denn das bedeutet dort nicht 100% das Gleiche.

Sobald das Interface zwischen 2 Komponenten in der Software feststeht können Entwickler unabhängig voneinander entwickeln, ohne dass dauernd neu abgesprochen werden muss. So kann sehr viel sicherer entwickelt werden und unerwünschte Effekte können vermieden werden.

Wie Tom DeMarco schon sagte: "Die Definition der Interfaces ist das Wichtigste am Projekt". Ohne eine Definition geht jedes Projekt unter. Von ihm ist übrigens auch das Mannjahr-Beispiel.

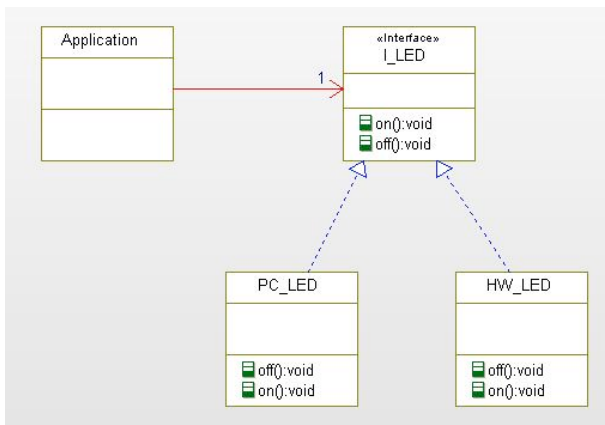


Ein weiteres Argument für die Aufteilung in Komponenten liegt in dem immer größer werdenden Druck, Projekte schneller ausliefern zu müssen. Auch genannt "Time to Market". Hierdurch ist immer weniger Zeit zum Entwickeln übrig und die Entwickler sind gezwungen, die Software so aufzubauen, dass wenigstens Teile davon wieder verwendet werden können.

Die Wichtigkeit einer richtigen Schnittstellenbeschreibung muss hier nicht extra betont werden, oder? ;)

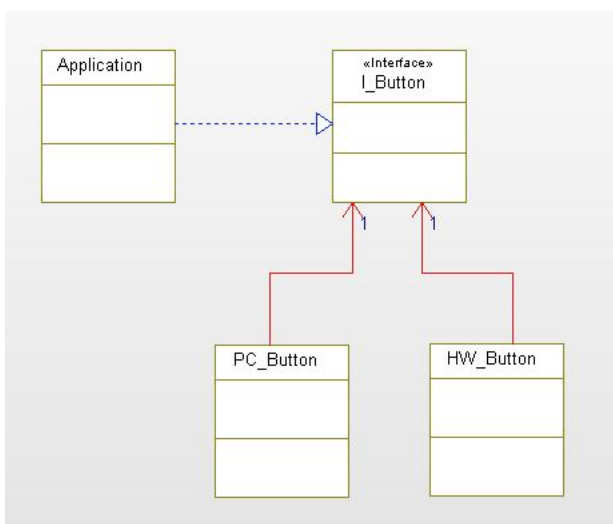
Was ist denn eine richtige Schnittstellenbeschreibung? In der Software besteht sie aus einer Beschreibung von Funktionsaufrufen mit Parameter- und Rückgabewerten. Dazu gehört eine Beschreibung, was diese Funktion genau macht, oder besser, was in ihrer Verantwortlichkeit liegt. Ein Funktionsaufruf ist eine Form von synchroner Kommunikation (In der UML gibt es neben synchroner auch asynchrone Kommunikation). Diese besteht aus einer Beschreibung von Events und eventuell deren Parametern, die eine Komponente empfangen kann. Hört sich alles plausibel an, unterscheidet sich aber meistens gravierend zu den Funktionsaufrufen.

Mit Vererbung



Um das zu verdeutlichen betrachten wir oben stehendes Diagramm. Dort wird eine Interface-Definition für eine LED dargestellt. Diese wird von der Klasse "Application" verwendet. Eine einfache Darstellung der Wirklichkeit. Die Klasse Application weiß jetzt, weil sie das Interface I_LED kennt, was eine LED alles kann. Es gibt zwei public Funktionen, on() und off(), die durch andere Klassen aufgerufen werden können (deswegen public).

Von dem I_LED Interface gibt es zwei mögliche Implementierungen, eine für PC und eine für meine Hardware.



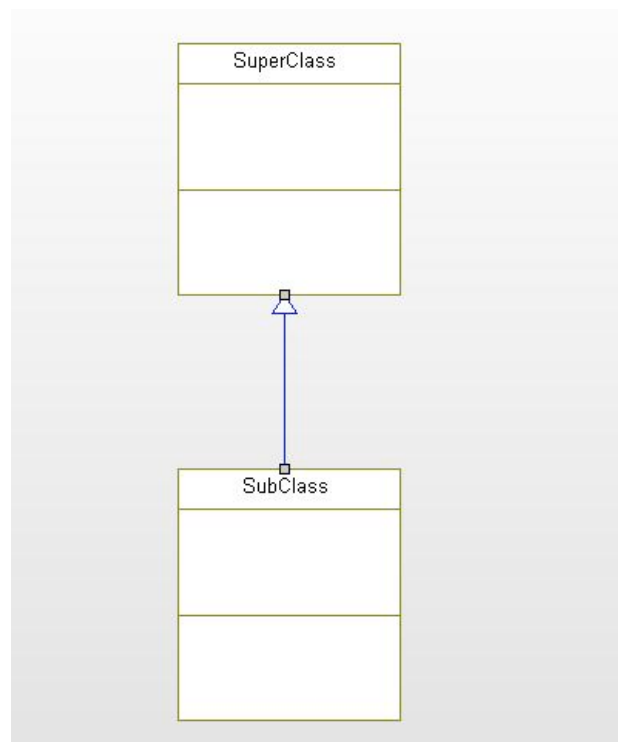
Der Applikation ist das alles egal, sie ruft nur Funktionen aus der Schnittstelle auf, welche dann ihre Aufgaben erfüllen. Eigentlich wäre das auch ohne Vererbung gut implementierbar, später schauen wir uns einen kleinen Trick an, wie das sehr elegant geht.

Die LED Klasse sollte hier eine wiederverwendbare Komponente sein. In der Art und Weise eine schöne Lösung. Anders wird die Situation, wenn eine Komponente aktiv ist, das heißt, sie sendet aus eigener Entscheidung Events, wenn bestimmte Ereignisse auftreten.

Das Problem hierbei ist, dass die zu schickenden Events, und der Empfänger von diesen Events, bekannt sein müssen. Das heißt, die Beziehung muss auch in die andere Richtung laufen und dafür braucht man die Vererbung, um das auch übersichtlich zu implementieren. (Siehe vorheriges Bild)

Dies alles war bis vor Kurzem in Rhapsody C gar nicht möglich, sondern nur in C++ oder Java. Glücklicherweise besitzt Rhapsody seit der Version 7.0 auch in 'C' die Möglichkeit, Code für eine Vererbung von Interfaces zu generieren. Eine schöne und elegante Methode, um Schnittstellen zu beschreiben; es gibt aber auch andere Methoden, die manchmal große Vorteile bieten.

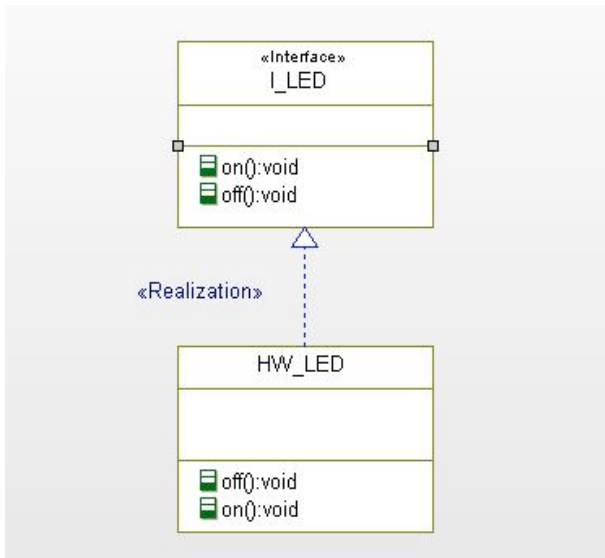
Rhapsody in C bietet keine vollständige Vererbung, nur von Interfaces kann vererbt werden.



Vererbung von Klassen kann dargestellt werden, dient aber nur zu Dokumentationszwecken. Beim Code generieren kommt eine Warnung, dass die Vererbung ignoriert wird. Das ist aber nicht so tragisch, da die Vererbung von Interfaces uns alleine schon ganz viele Vorteile bietet.

Wie funktioniert es?

Um das heraus zu finden nehmen wir ein kleines Beispiel und schauen uns den generierten Code an.



I_LED definiert, dass eine LED zwei Funktionen hat, on() und off(). In Rhapsody in C haben public Funktionen ein Präfix, den Namen der Klasse. Der 'C' Compiler kann sonst doppelt vergebene Namen nicht auflösen. Die Funktionsaufrufe sehen wie folgt aus:

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

Der generierte Code implementiert das mit Hilfe einer Funktionspointertabelle, die diesen Aufruf mit der richtigen Funktion verknüpft. Die Deklaration der Klasse sieht so aus:

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

Die Vtbl wird in der Init Funktion (Constructor) mit den richtigen Funktionspointern gefüllt.

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

Der Aufruf einer der Funktionen beinhaltet, dass die richtige, aufzurufende Funktion ermittelt und ein korrekter "me" pointer berechnet wird. Abhängig davon, was eine Klasse macht (Active, statechart, monitor) zeigt der "me" pointer nicht immer auf die gleiche Stelle.

```
/**# 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);
        }
    }
}
```

Schauen wir jetzt auf die Implementierung von HW_LED. Die Struktur beinhaltet eine Instanz von der I_LED Klasse.

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

Gefüllt wird diese Struktur in der Init Funktion

```
/**# 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);
}
```

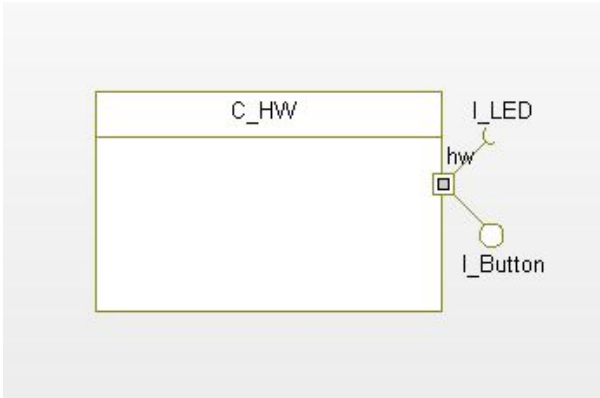
Die Deklaration der Vtbl steht statisch in dem Konstruktor. Alle notwendigen Pointer und Offsets werden hier direkt festgelegt.

Der wirkliche Inhalt der off() und on() Funktionen steht in dem HW_LED_on() und HW_LED_off() body.

Fazit: Es kostet ein bisschen Code und Laufzeit, ist dafür aber sehr übersichtlich.

Ports, das geht auch.

Eine andere Methode, um Schnittstellen zu implementieren sind Ports. Seit der UML 2.0 sind Ports ein Teil der UML. Wenn eine Klasse kommuniziert, kann sie einen Port haben, welcher mit den Interfaces verbunden ist. Es gibt zwei Arten von Interfaces: Required (erforderlich) und Provided (vorausgesetzt). Genauso wie in unserem vorherigen Beispiel mit den Interfaces. Auch Rhapsody in 'C' unterstützt Ports. Da Teile davon im Framework gelöst sind, muss auch das Framework die Ports unterstützen. Mittlerweile ist das bei allen Willert Frameworks der Fall.



Ports sind eigentlich nur eine nettere Schreibweise unserer Interface Methode. Sie unterscheiden sich aber doch sehr im Vergleich zu den Interfaces: Zum Beispiel werden Ports erst zur run-time verbunden. Diese Methode nimmt allerdings Platz und Zeit weg und ist auch ziemlich komplex. Bei Interfaces muss der User selbst eine Verbindung zwischen den kommunizierenden Objekten erstellen. Ports machen dies automatisch. Ein Event oder ein Funktionsaufruf findet ja auch nicht in einem anderen Objekt, sondern im Port selbst statt.

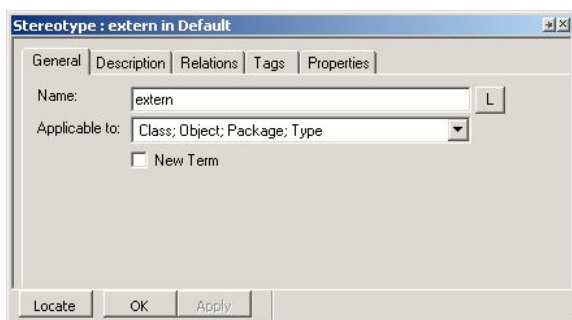
```

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

```

Im Framework befindet sich Code, der dafür sorgt, dass das Event an der richtigen Stelle ankommt.

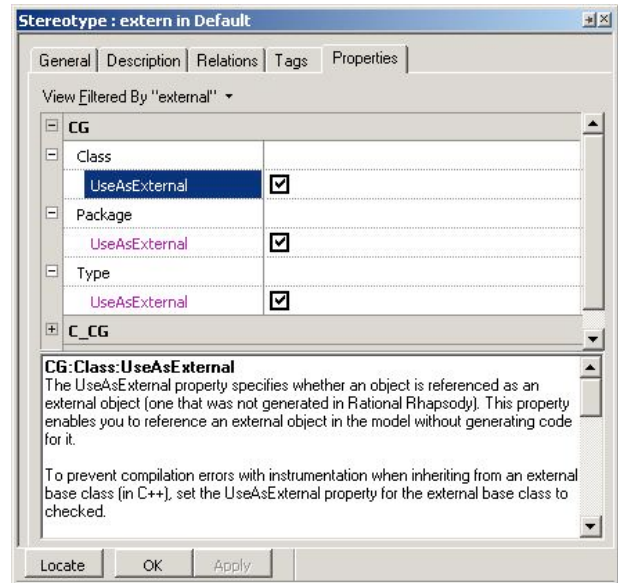
Das macht Ports weniger geeignet für harte Echtzeit-Anwendungen und dort, wo es um das letzte Byte geht. Die derzeitige Implementierung verwendet außerdem noch starke Rekursion, was eine Zertifizierung nach SIL oder DAL sicherlich nicht einfacher macht. Die Darstellung ist aber sehr verständlich.



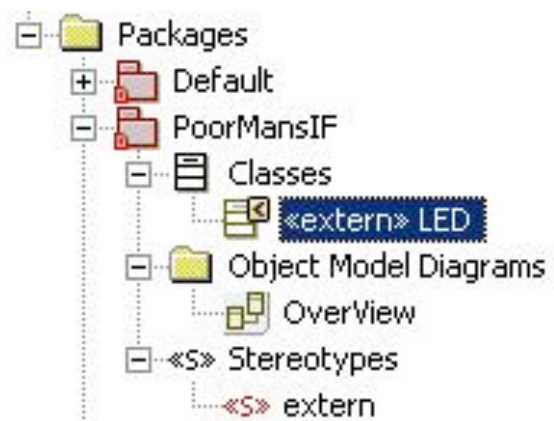
Extern: Einfach und gut

Brauchen wir das denn unbedingt? Die Antwort ist kurz und klar: Nein. Es geht einfacher, vielleicht grafisch weniger elegant, dafür aber mit geringerer Codegröße und weniger Komplexität.

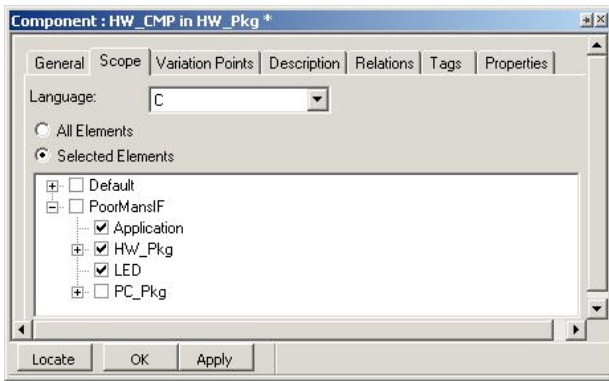
Es funktioniert mit einem kleinen Trick. Rhapsody kann den Elementen Class, Object, Type und Package das Property "UseAsExternal" mitgeben. Das heißt dann, dass dieses Element da ist, aber dass Rhapsody dafür kein Code generieren soll. Um das ganze sichtbar zu machen, sollte man einen Stereotype erstellen, welcher dieses Property setzt.



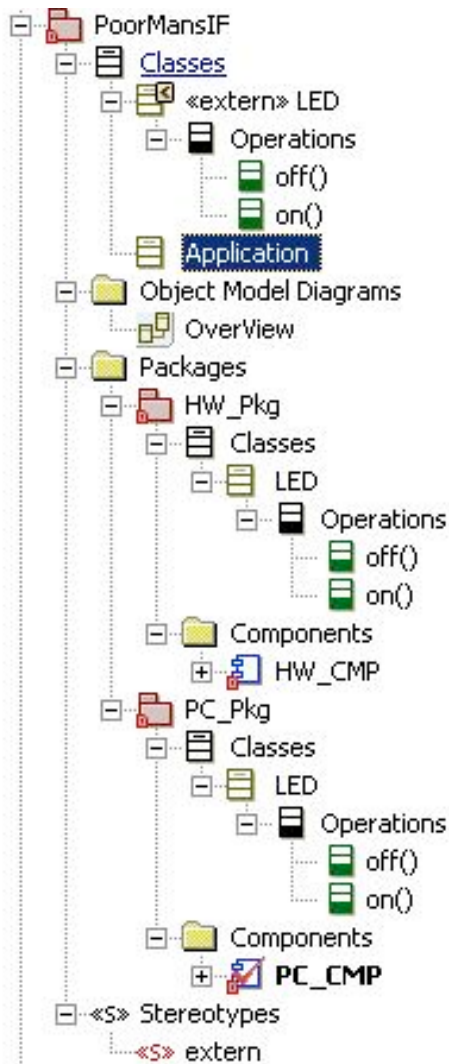
Wenn dieser Stereotype einem Element zugewiesen wird, erscheint im Browser ein kleines gelbes Symbol, welches bedeutet, dass es ein externes Element ist.



Wie gesagt, Rhapsody generiert für dieses Element kein Code. Es geht davon aus, dass der Anwender sich selbst darum kümmert. Hier greift unser Trick: Wir sorgen dafür, dass in unserem Modell an einer anderen Stelle für jede Implementierung ein weiteres Element "LED" vorhanden ist. Mit Hilfe von "Scope" können wir in unserer Hardware-Komponente einstellen, dass in jeder Hardware-Komponente genau einmal unsere LED vorkommt. Allerdings mit einer anderen Implementierung.



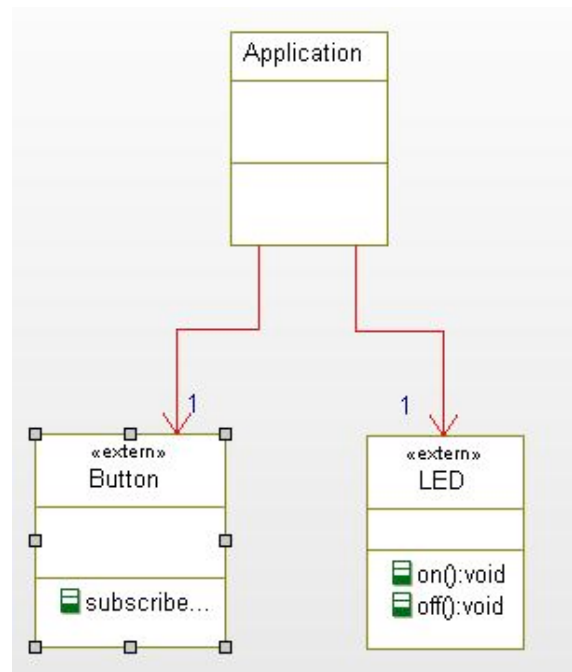
Da die gleiche Klasse in einem Package nicht zweimal auftauchen kann, benötigen wir für unseren Trick mehrere Packages. Der Nachteil ist, dass die Elemente genau den gleichen Namen haben müssen, sonst funktioniert es nicht. Auch kann die Option, dass Rhapsody für jedes Package ein eigenes Verzeichnis generiert, nicht verwendet werden. Sonst wird das include file LED.h nicht gefunden.



Hier ist eine Übersicht, wie diese Lösung im Rhapsody Browser aussieht. Die Applikation kann eine Beziehung mit unserer externen LED-Klasse haben und ihre Funktionen aufrufen. Nach dem Generieren und Kompilieren wird dann automatisch die richtige aufgerufen.

Publisher Subscriber

Das vorangegangene Beispiel funktioniert wunderbar und erzeugt kein Byte extra Code. Leider ist es nicht ohne weiteres anwendbar, wenn Events verschickt werden müssen. Dann tritt das Gleiche auf, wie bei den Interfaces. Es sind fest programmierte Namen der Klassen bekannt, die als Software-Komponente verwendet werden müssen. Bei Interfaces können wir die Beziehungsrichtung umdrehen. Hier geht das nicht so leicht. Die Application-Klasse können wir logischerweise nicht so einfach extern setzen. Aber es gibt dafür einen Mechanismus, der auch in 'C' als callback Funktion bekannt ist. Es ist eigentlich ein so genanntes Design Pattern, welches wir als Publisher Subscriber kennen (Verleger-Abonnent)



Die Klasse, die Events verschicken kann, bietet eine Subscribe Funktion an, wo jeder (der Einfachheit zuliebe nur einer) sich als Empfänger dieser Events anmelden kann. Das geschieht durch das Weitergeben von einem Pointer auf eine private Funktion, die einem das Event zuschickt.

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

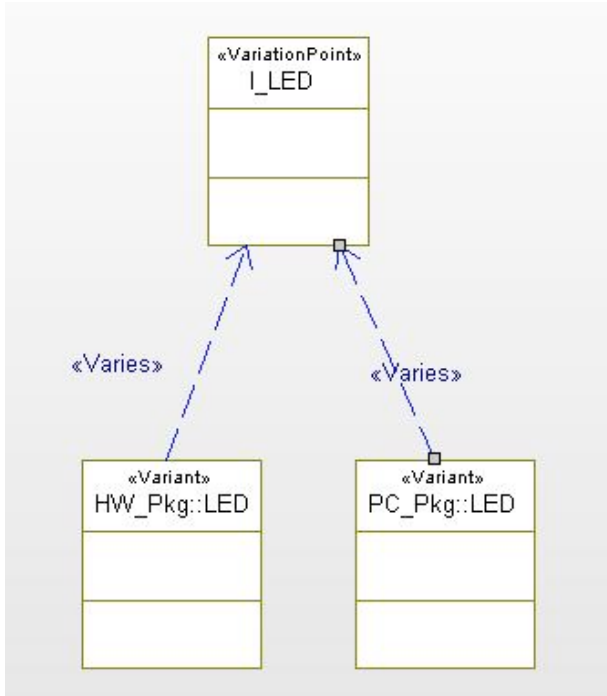
Zusätzlich übergibt der Aufrufer ein Pointer auf sich selbst (ein "me" Pointer). Den brauchen wir im Aufruf der Funktion. Die Publisher Klasse speichert die Parameter und sorgt dann dafür, dass diese Funktion aufgerufen wird, sobald ein Ereignis auftritt.

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

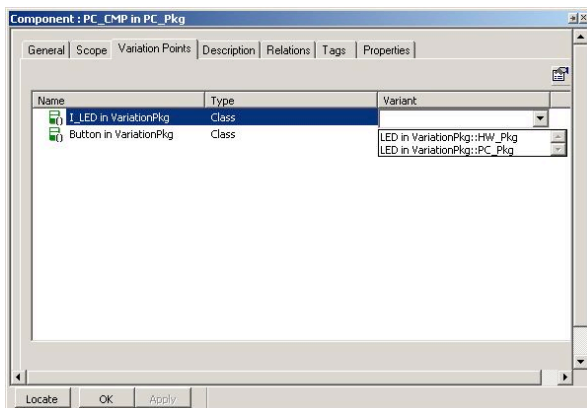
Auf diese Weise kennt die Application unsere Hardware-Komponente, die selbst kein Wissen über die Anwendung braucht.

Variation Points

Seit der Version 7.5 von Rhapsody ist der vorherige "Trick" formalisiert worden und als Feature verfügbar. Es handelt sich um die "Variation Points" welche mit Hilfe von mitgelieferten Stereotypes implementiert werden.



Das Interface wird jetzt mit Hilfe eines Stereotypes als "Variation-Point" definiert. Alle Implementierungen haben mit diesem Variation-Point eine Dependency-Beziehung (mit dem Stereotype "Varies"). Ausserdem haben die Implementierungen den Stereotype "Variant". Da sagt Rhapsody, dass es eine Implementierung vom Interface ist. In der jeweiligen Komponente legt man fest, welche von den Varianten man für diese Komponente nimmt.



Rhapsody generiert hierfür ein spezielles include file `I_LED.h`, das für alle Funktionsaufrufe Makros erstellt, die dann die richtige Funktion aufrufen. Der Unterschied zu der "externen Methode" ist, dass hier die Namen unterschiedlich sein müssen sonst kompiliert es nicht. Packages in Unterverzeichnissen zu generieren würde auch funktionieren.

Bei Klassen, die Events verschicken, könnte auch die Publisher Subscriber Methode verwendet werden, auf die gleiche Art und Weise, wie bei dem "extern" Beispiel.

Fazit

Was sollte man verwenden? Wie immer ist die Antwort abhängig von vielen Faktoren. Wenn keine Zertifizierung nach 61508, DO178b (oder anderen) ansteht kann man sich nach Belieben etwas aussuchen. Ports und Interfaces sind dabei sehr elegant und erhöhen die Verstehbarkeit des Modells enorm.



Wenn zertifiziert werden muss oder der Code MISRA konform sein soll oder wenn Speicher/ Laufzeit einen Engpass formen, wird man wahrscheinlich auf Ports und Interfaces verzichten. Bei sehr hohen SIL oder DAL Levels muss sogar auf Variation Points verzichtet werden, da diese Preprozessor Makros verwenden. Dann bleibt nur die "poor mans" Variante übrig. Aber auch die ist leichter zu verstehen, als ein Modell, bei dem alles direkt implementiert ist.

Wie begegnet man steigender Komplexität in der Software-Entwicklung am effizientesten?

Mit dem Embedded UML Studio II™ ist Production Code - Generierung aus UML-Modellen so einfach, wie C-Programmierung mit Compiler IDE.



Glauben Sie uns nicht? Testen Sie es!
Evaluieren Sie einfach kostenlos unsere UML-Umgebung:
www.willert.de/uml-getting-started

Walter van der Heiden

„LIVE“ ZU DIESEM THEMA

auf folgender Veranstaltung:

2010-12-09 ESE Kongress in Sindelfingen

<http://www.esk-kongress.de>

Weitere Ausgaben unserer News- oder Techletter finden Sie unter: www.willert.de/Newsletter



Video-Interview

Pressemitteilungen

■ **News- und Techletter**

News- oder Techletter abonnieren

News- und Techletter

Hier können Sie unsere News- oder Techletter abonnieren

- **Newsletter No 24** Richtiges richtig tun
- **Techletter No 1** Interfaces in Rhapsody in C

- **Newsletter No 23** Der steinige Weg zu fehlerfreier Software
- **Newsletter No 22** Komplexität managen - modellieren statt programmieren
- **Newsletter No 21** Real-Time Architektur Design - jonglieren mit der Zeit
- **Newsletter No 20** ARM-Cortex-Architektur in Embedded Systemen
- **Newsletter No 19** Heutzige Anforderungen an effizientes Software Engineering
- **Newsletter No 18** Gutes Werkzeug ist halbe Arbeit
- **Newsletter No 17** Embedded Software Development

Hier können Sie die News- oder Techletter abbestellen



Herausgeber:

WILLERT SOFTWARE TOOLS GMBH

Hannoversche Straße 2 |

31675 Bückeburg

Tel.: 05722 - 9678 60

www.willert.de

info@willert.de

Autor: Walter van der Heiden