EVOCEAN
guiding you beyond

UML statt „C“

Vorurteile gegenüber der UML – Fiktion oder Realität ?

Einleitung

Obwohl die UML mittlerweilen über 15 Jahre alt ist, ist ihre Anwendung im Embedded Real-Time Bereich immer noch dürftig. Aber gerade für den Embedded Bereich, welcher mit sehr hohen Anforderungen an Sicherheit, Zuverlässigkeit und Echtzeit verbunden ist, wäre der Einsatz von UML gewinnbringend.

Wir zweigen einige der großen Vorteile auf, welche UML bezüglich Analyse, Design, Implementierung, Dokumentation, Kommunikation, Test und Architektur bietet – im Vergleich zur herkömmlichen Entwicklung in „C“. Gleichzeitig zeigen wir Ihnen anhand eines Beispiels mit IBM Rational Rhapsody, wie der UML Ansatz in der Praxis implementiert werden kann.

Nutzen und Besonderheiten

Wenn

- noch in „C“ entwickelt wird
- Projekte zu spät, nicht komplett oder gar nicht abgeschlossen werden
- 90% der Zeit getestet wird
- die Dokumentation nicht vorhanden oder auf einem anderen Stand ist, als die dazugehörende Anwendung
- Mitarbeiter gestresst herumlaufen
- Emulatoren tagtäglich benutzt werden müssen
- Kaffee und Pizza's Standard-Essen sind
- die Software Qualität zu niedrig ist
- Änderungen in der Software unmöglich sind…

… dann sind Sie reif für eine andere Art der Software-Entwicklung, basierend auf dem Stand der heutigen Technik. Denn mit Hilfe von UML können viele oben genannten Probleme gelöst werden.

Wir zeigen Ihnen, wie Sie die allgemein bestehenden Vorurteile gegenüber UML getrost vergessen können.

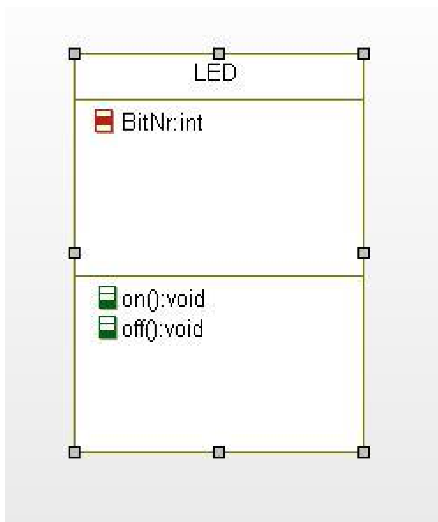
Vorurteil 1
UML ist nicht geeignet, weil wir kein C++ verwenden

Richtig – UML ist eine objektorientierte OO-Sprache. OO ist aber nicht an eine Sprache gebunden! OO kann man in jeder Sprache verwenden, sogar in Assembler. Es gibt zwei Regeln, die man dabei einhalten sollte:

1. Erstelle OO-Konventionen zum Abbilden in einer Sprache
2. Halte dich daran!

Die zweite Regel ist die Schwierigere.

Klassen sind leicht als Strukturen abzubilden. In C++ sind auch Funktionen Teil der Klasse. Das geht in „C" auch, sofern wir mit Funktionspointer arbeiten. Eine bessere Lösung ist jedoch, die Funktionen mit einer Namens-Konvention in die Klasse einzubinden. D.h. eine Klasse LED mit Attribut BitNr und Funktionen on() und off() kann man in „C" als Funktionen Led_on() und LED_off() abbilden, wie hier im Bild beschrieben:




```
typedef struct LED
{
   int BitNr;
};
LED *LED_Create( int aBitNr )
{
   LED *me = malloc( sizeof( LED ) );
   LED_Init( me, aBitNr );
}
void LED_Destroy( LED *me )
{
   LED_Cleanup( me );
   free( me );
}
void LED_Init( LED *me, int aBitNr )
{
   if( me != NULL)
   {
      IODIR1 |= 1 << aBitNr;
      me->BitNr = aBitNr;
   }
}
void LED_Cleanup( LED *me )
{
      // Do nothing
}
void LED_on( LED *me )
{
   IOSET1 = 1 << me->BitNr;
}
void LED_off( LED *me )
{
   IOCLR1 = 1 << me->BitNr;
}
```

Eine Klasse hat einen Constructor (Destructor), der beim Erstellen (Vernichten) eines Objektes automatisch aufgerufen wird. Das geht nicht in „C"! Deswegen bekommt jede Klasse ein eigenes new() und delete() an der Stelle, an welcher der Aufruf von con- und destructor gemacht wird.

In einer OO-Sprache wird bei einem Funktionsaufruf automatisch das richtige Attribut durch den Compiler ausgewählt. Oder man verwendet den „this" pointer. Auch das funktioniert in „C" nicht; wir geben jeder Funktion einen „me" pointer, welcher auf das aktuelle Objekt zeigt.

In C++ oder Java würde eine Funktion so aufgerufen: Objekt->Funktion().
In „C" funktioniert es so: Klasse_Funktion(Objekt). Etwas anders, aber sehr ähnlich.

Regel „Zwei" ist viel einfacher zu handhaben, wenn ich meinen Code in zwei Dateien – eine für jede Klasse – generiere: <klasse>.c und <klasse>.h.

Fazit:
Die Verwendung von „C" als Basissprache ist keineswegs ein Problem beim Einsatz von UML.

Vorurteil 2
Generierter Code ist nicht zertifizierbar

Eine Zertifizierung ist ein Prozess, der eigentlich nicht vom Code abhängig ist. Das gilt für alle Arten von Zertifizierungen; 61508, 50128, 26262 oder DO178, gleichgültig nach welchem SIL- oder DAL-Level.

Was zertifiziert wird, sind die Beschreibungen von Anforderungen, ob und wie diese implementiert und getestet wurden (richtige Implementierung der Anforderungen).

Das trifft sowohl auf handgeschriebenen Code wie auch auf automatisch generierten Code zu. Der automatisch generierte Code hat jedoch einen großen Vorteil in diesem Prozess: Er hat eine Verbindung mit den Anforderungen, aus welchen er erstellt wurde. In UML können Anforderungen mit Modell-Elementen verknüpft werden, woraus später ein Code generiert wird.

Auch sorgt die automatische Code Generierung dafür, dass der Code viel konsistenter aufgebaut ist.

Gleichzeitig ist das UML-Modell die Dokumentation, die man in einer Zertifizierung braucht! Der Code ist quasi ein – sehr sinnvolles – Abfallprodukt.

Natürlich wird eine Zertifizierung nicht stattfinden, ohne dass dabei ein Blick auf den Code geworfen wird. Das wird jedoch nur anhand von Stichproben gemacht um festzustellen, ob handwerklich sauber gearbeitet wurde; z.B. Einhaltung von Codierungsrichtlinien. Bei den meisten UML-Werkzeugen, welche Code generieren, kann man das „Wie" der Code Generierung weitgehend einstellen. Ferner können Custom Model Checks dafür sorgen, dass auch der handgeschriebene Teil eines Codes allen Richtlinien entspricht.

Wenn der Code aus einem UML-Tool generiert wird, sollte sichergestellt werden, dass der Anforderungstext mit integriert wird. Damit diese Information immer verfügbar ist. Da es sich um generierten Text handelt, wird dieser automatisch mit dem UML-Modell synchronisiert.

Bei einigen Tools ist es möglich, externe Änderungen im Code automatisch wieder ins Modell zurück zu führen, was die Modell-Code-Verbindung nochmals verbessert.

UML Tools verwenden oft ein Framework, um fehlende Funktionalität in „C" zu implementieren. In UML kann man sich auf einem höheren Level ausdrücken als in „C". Es gibt zum Beispiel Elemente, um Threads, Timer oder Events darzustellen. Das alles gibt es in „C" nicht.

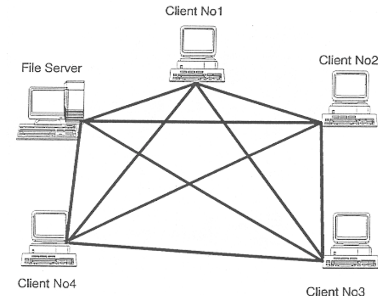
Das Framework muss in einer zertifizierten Anwendung ebenfalls zertifiziert werden. Das sieht auf den ersten Blick nach Extra-Aufwand aus. Betrachtet wir jedoch mal genauer, was geschieht, wenn kein Framework und generierter Code eingesetzt wird: Dann wird ein eigenes Laufzeitsystem programmiert, welches auch zertifiziert werden muss. Vor allem Laufzeitsystem welche auf einem mainloop basieren haben einen Code, der sich durch die gesamte Anwendung zieht… was eine Beschreibung nicht erleichtert!

Fazit: Die Zertifizierung von generiertem Code ist möglich! Generieren von Code hat zwar auch Nachteile, die vielen Vorteile überwiegen jedoch bei Weitem.

Vorurteil 3
UML eignet sich nur für große Projekte mit viel Speicherplatz und ohne Echtzeit-Anforderungen

Natürlich ist die UML für große Projekte sehr geeignet. Das bedeutet aber nicht, dass sie nicht auch für andere Umgebungen geeignet ist. Der Unterschied ist, dass man sich nicht um Code-Größe oder Laufzeit kümmern muss!

Software ist in den letzten Jahren massiv komplexer geworden. Das kommt, unter Anderem, von der zunehmenden Vernetzung. Sogar kleine Programme beinhalten Teile, die mit verschiedenen anderen Komponenten kommunizieren müssen. Je mehr Netzwerk-Knoten man hat, desto mehr Verbindungen entstehen. Diese Beziehung ist nicht linear; 1 Knoten hat 0 Verbindungen, 2 Knoten haben 1 Verbindung, 3 - 3, 4 - 6, 5 - 10, 6 - 15 usw. (für Mathe-Fans die Formel: N(N-1)/2).



Dazu kommt, dass es häufig viele Software-Varianten gibt, welche die Komplexität noch zusätzlich erhöhen. Das hat nichts mit Code-Größe zu tun! Im Gegenteil; wenn der Code in einen Speicher im Kilobyte-Bereich passen muss oder wenn Millisekunden bereits eine lange Zeit sind, dann macht der Einsatz von UML vieles einfacher und übersichtlicher.

Aber generierter Code ist doch viel größer und langsamer als handgeschriebener Code!? Und es kommt ja auch noch ein Framework dazu. Passt doch nie im Leben!

Dies stimmt nicht unbedingt. Klar, ein „Hello World“ wird wahrscheinlich etwas größer, wenn es aus UML generiert wird. Aber sobald wir von etwas grösseren Anwendungen sprechen kommen zwei Effekte zum tragen: „Toter Code“ und „ineffizienter Code“.

Wenn ein Zustands-Diagramm per Hand kodiert wird, ist das nicht zwangsläufig ineffizient. Aber wenn sie nachträglich geändert werden muss ist es unvermeidbar, dass „ineffizienter Code“ rauskommt. Das macht generierter Code erheblich besser.

„Toter“ oder, noch schlimmer, „halb-toter Code“ sind Funktionen, die mal programmiert wurden und niemand mehr weiss, ob sie doch noch verwendet werden oder nicht. Das kostet Speicherplatz! Wenn dann dieser Code auch noch zufällig Puffer schafft um z.B. Array Überläufe von anderen Funktionen aufzufangen, findet man ihn bei den Tests nicht mehr. Solchen Code findet man meistens nur, wenn der Code mit statischen Analysen beurteilt wird.

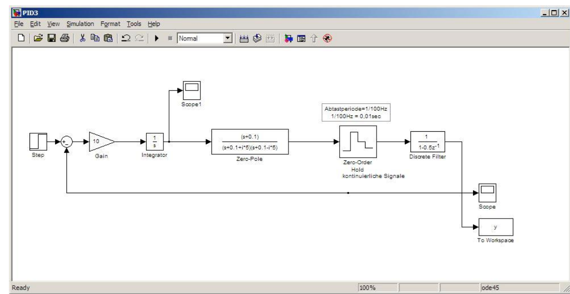
Das alles passiert ab einer bestimmten Code-Größe und eigentlich auch ab einem bestimmten Projekt-Alter, wenn der generierte Code kleiner ist als der handgeschriebene. Die Grenzen liegen schon bei 3-stelligen Kilobyte-Bereichen und bei Projekten, die über ein Jahr dauern.

Code Generatoren werden auch immer effizienter. Vergleiche Sie mal C-Compiler (die eigentlich Assembler Code-Generatoren sind) mit handgeschriebenem Assembler-Code; sogar der beste Assembler- Programmierer verliert da das Wettrennen gegen die Maschine!

Dazu kommt, dass, was die Laufzeit und Code-Menge anbelangt, auf den Code eines Programms meistens die 80/20 Regel anwendbar ist. Das bedeutet, dass 80% des Codes nur 20% von der Laufzeit, die restlichen 20% Code die jedoch 80% der Laufzeit konsumieren. Die meisten Embedded Anwendungen verbringen 80% ihrer Zeit in der Treiber Schicht, in welcher aber nicht die Komplexität der Anwendung liegt! Diese liegt eher in einer anspruchsvollen Menüsteuerung oder in der Beurteilung von Prioritäten, wenn mehrere Busse gleichzeitig Kommandos schicken können.

Fazit: UML ist sehr gut geeignet, die zunehmende Komplexität von Anwendungen zu meistern. Code- Größe oder Echtzeit haben darauf kein Einfluss.

Vorurteil 4
UML ist für mein Art von Projekten nicht geeignet



Die UML wurde eigentlich nicht für eine spezifische Branche oder Projekt-Art entworfen. Die UML wurde gemacht, um Software modulieren zu können. Da sie sprachenunabhängig ist ergeben sich keine Nachteile mit verschiedenen Programmiersprachen.

Die „nicht geeignet“-Aussage kommt oft aus Projekten, die stark auf synchronen Architekturen basieren – z.B. Automotive-Projekte. Häufig wird in solchen Projekte sogar modelliert, aber nur mit proprietären Tools wie beispielsweise Matlab Simulink.

Die UML ist ein reines Software-Modellierungswerkzeug! Deswegen enthält sie keine Diagramme um z.B. Regler zu programmieren. Es gibt jedoch Diagramme, um Verhalten zu definieren und zu programmieren. So zum Beispiel Zustands-Diagramme (eigenen sich weniger für die Darstellung von Reglern) oder aber Aktivitäts-Diagramme, welche dafür bestens geeignet sind.

Die UML hat Diagramme, die Architektur, Kommunikation und Verhalten darstellen. Somit kann man jede Anwendung modellieren, es gibt praktisch keine Grenzen.

Das Verhalten von mit UML modellierten Anwendungen ist nicht auf zeitdiskretes Verhalten limitiert. Man kann auch zeitkontinuierliches Verhalten modellieren! Das Problem dabei ist, dass oft versucht wird, dieses mit einem Zustands-Diagramm zu modellieren. Die Timer, die man in Zustands-Diagramme verwendet, sind jedoch sehr ungenau und auch nicht deterministisch: sie sind dafür nicht geeignet. Zeitkontinuierliches Verhalten soll mit Aktivitäts-Diagrammen modelliert werden, die dann durch das UML-Tool auf die richtige Art und Weise in die unterliegende Laufzeitarchitektur integriert werden müssen.

Diese Laufzeitarchitektur muss natürlich geeignet sein, die Anwendungsanforderungen zu erfüllen. Wenn eine Reaktionszeit von Mikrosekunden nötig ist sollten die Hardware und das Betriebssystem das auch unterstützen. Das hat aber mit UML nichts zu tun. Selbst wenn das verwendete UML-Tool nicht die entsprechende Codegenerierung besitzt, kann man trotzdem alles modellieren und den Code anderweitig einbauen.

Sogar Interrupts lassen sich mit der UML modellieren. Viele Tools unterstützen das Einbinden externer Code in ein UML-Modell. Sogar Reverse Engineering ist möglich, so dass existierender Code modelliert werden kann. Das ist oft notwendig, um bestimmte typische Hardware unterstützen zu können.

Die UML ist auch Domänen-spezifisch anpassbar. Mit Hilfe von ‚stereotypes‘ und ‚tags‘ können die Eigenschaften von UML-Elementen an konkrete Anforderungen der aktuellen Domäne angepasst werden.

Viel hängt natürlich von dem verwendeten Framework ab. Wenn ein Framework eingesetzt wird, welches die erforderliche Software-Architektur maximal unterstützt, kann die UML alle Anforderungen der Modellierung erfüllen.

Neben der UML gibt es noch die SysML, die Systems Modelling Language. Ein Profil, welches stark auf UML basiert, sich jedoch auch für die Modellierung von Systemen eignet. Dass heisst, dass auch Hardware beschrieben werden kann, was für embedded Anwendungen oft notwendig ist. Denn häufig wird die Hardware zusammen mit der Software entwickelt.

Fazit: Die UML kann für alle Arten von Projekten eingesetzt werden, nicht nur für zeitdiskrete Umgebungen.

Vorurteil 5
UML ist komplex – handgeschriebener Code einfacher

Das ist, zumindest teilweise, wahr. UML ist nicht einfach zu lernen. Genau so wenig wie „C“! Viele „C“ Programmierer haben einfach verdrängt, wie sehr sie bei ihren ersten „C“ Code geschwitzt haben. Ein erfahrener „C“ Programmierer wird man auch nicht nach ein paar Tagen; das dauert Jahre. Das Erlernen der UML braucht auch seine Zeit!

Es ist so gut wie unmöglich, UML autodidaktisch zu lernen. Im Speziellen wenn man ein Tool mit Codegenerierung einsetzt. Es gibt heutzutage jedoch sehr gute Seminare und Kurse. Berücksichtigen soll man bei der Auswahl der Ausbildung, dass es viele Kurse gibt, die die UML zwar theoretisch erklären, aber nicht sehr praxisorientiert sind oder „nur Bilder malen.“

UML ohne Codegenerierung einzusetzen ist wie man „C“ aufschreibt und später daraus selber einen Assembler macht. Der Lerneffekt ist dabei = 0 und es gibt auch keine automatische Beziehung zwischen dem „C“-Modell und dem Assembler Code. Man wird also nie erfahren, ob man richtiges „C“ geschrieben hat! Das Gleiche trifft auf UML zu: die richtige Syntax und Semantik von UML lernt man erst, wenn Codegenerierung eingesetzt wird. Denn diese gibt das Feedback, ob das Modell richtig ist oder nicht.

UML ermöglicht einem die Modellierung einer wesentlich höheren Komplexität als herkömmliche Programmiersprachen. Und genau in diesem Punkt erweist sich obiges Vorurteil als falsch. Handgeschriebener Code ist zwar einfacher zu erlernen… aber in komplexen Projekten definitiv nicht einfacher anzuwenden. Technologien haben eine maximale Komplexität, welche sie beherrschen können. Versuchen Sie mal, Windows Programme mit Lochkarten in Assembler zu erstellen! Dann verstehen Sie, was hier gemeint ist.

Dass es mehr Zeit braucht, um UML als „C“ oder Assembler zu lernen, stimmt. Jedoch sind die Applikationen, welche wir mit UML erstellen können, auch um ein Vielfaches komplexer als diejenigen, welche mit anderen Hochsprachen erstellt werden können. Und alle Technologien, welche das Handling weiterer Komplexität ermöglichen, sind lernintensiver!

UML wird jedoch derzeit an vielen Hochschulen und Universitäten bereits unterrichtet. Immer mehr Absolventen beherrschen UML und die Zahl der Entwickler, die schon Erfahrung mit der UML und/oder UML Tools haben, nimmt kontinuierlich zu.

Fazit: UML ist zwar komplexer als „C“, erlaubt aber eine viel höhere Komplexität der modellierten Anwendungen.

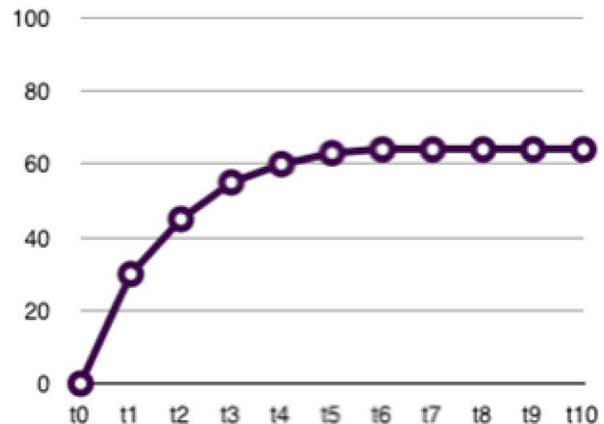
Vorurteil 6
Für mehr Qualität braucht man nicht UML sondern mehr Tests

Dem stellen wir zuerst einmal eine andere, weiterverbreitete Aussage gegenüber: „Qualität kann man nur rein designen, nicht rein testen.“

Egal wie oft man seine Anwendung testet; Fehler sind immer noch vorhanden. Mit jeder Methodik finden Sie nur eine gewisse Klasse von Fehlern – eine Fehler-find-Quote von 100% ist unmöglich.

Fehlersuche ist ein Prozess, in dem das Ergebnis eine e-Kurve beschreibt: je länger ich suche, desto weniger (neue) Fehler finde ich.

Angenommen, man findet 65% von seinen Fehlern nach dem ersten Testen. Das würde bedeuten, dass bei anfänglich 1000 Fehlern man bereits 650 Fehler gefunden hätte. Das heisst aber auch, dass noch 350 Fehler in der Anwendung sind. Wenn man aber mit 100 Fehler angefangen hätte wären nur noch 35 Fehler übergeblieben. Daraus resultiert, dass bei einem besseren Design und einer besseren Entwicklung viel weniger Fehler vorhanden sind als bei einem weniger guten Design. In letztem Fall kann man fast unendlich weiter nach Fehler suchen.



Kann man mit UML die Qualität von Software erhöhen? Natürlich kann man das! Die UML ermöglicht eine viel bessere Übersicht der Anforderungen, der Analyse, des Designs und der Implementierung von Software. Auch bietet die UML viel mehr Kontrolle über die Laufzeit.

Selbst der beste Source-Code Editor hat keine Möglichkeit, den Anwender zu warnen, wenn ein fehlerhafter Code editiert wird. Die UML bietet viele Möglichkeiten, beim Modellieren oder spätestens beim Generieren von Source-Code auf inkonsistente Modelle hinzuweisen.

Software Entwickler, die nur in einem Editor arbeiten, kann man mit Architekten vergleichen, die nur mit einer Schreibmaschine anstelle eines CAD-Programms arbeiten. Da gibt es keine Übersicht über die vier verschiedenen Schichten im Programm: Zeit, Funktionalität, Daten und Priorität. Im „C“ Code kann jede Code Zeile Beziehungen zu diesen 4 Schichten enthalten. In einem UML-Modell sind viele Schichten durch den Einsatz von Objektorientierung getrennt. Außerdem hat man in einem UML-Modell jederzeit die Übersicht, welchen Einfluss jedes Modell-Element auf Zeit, Funktionalität, Daten oder Priorität hat.

Fazit: Testen ist gut, modellieren ist besser!