Objektorientierung in der Grafischen Programmierung mit Funktionsblöcken, 2020-08-06

Dr. Hartmut Schorrig, www.vishia.org

Objektorientierung in der Grafischen Programmierung mit Funktionsblöcken

Dieser Artikel wirft einige diskussionswürdigen Fragen der Grafikprogrammierung mit FBlocks im Zusammenhang mit der Objektorientierung auf

Table of Contents

Objektorientierung in der Grafischen Programmierung mit Funktionsblöcken, 2020-08-06

1. FBlock-Diagramme und Objektorientierung

Zunächst die Frage: **Was ist Objektorientierung**? Einfache Antwort – „***Verbindung von Daten mit den zugehörigen Operationen.***“ Nichts weiter. Operationen werden auch „Methoden“ genannt.

Fragen der Vererbung, Aggregation und Composition etc. sind zweitrangig und möglich.

Zweite Frage: **Trifft das auf FBlock-Diagramme zu?** Klare Antwort: ***Selbstverständlich***. Auf intrinsic-Weise enthalten die FBlocks Operationen, die mit den eigenen Daten arbeiten.

Die in einer Bibliothek typischerweise vorrätigen FBlocks sind Klassen, sie definieren Daten und Operationen mit einem Dateninterface. Die in ein Modell hineingezogenen Lib-FBlocks sind Objekte, inkarnierte Klassen. Die Daten dieser FBlock-Objekte sind im Modell-Modul angeordnet. Letztlich werden sie beim startup meist aus dem heap instanziiert.

Dritte Frage: **Wie ist es mit der üblichen Datenkapselung und Aufruf von Operationen?** Antwort: Auch geklärt. Die FBlocks haben Interndaten, die sich mit deren Programmierung ergeben (kann entweder wieder grafisch sein, oder in einer Zeilenprogrammiersprache). Da die FBlocks (meist) wiedereintrittsfähig sind werden die Interndaten mit dem Objekt angelegt. Das ist private-Kapselung.

Die Daten an den Connections sind nicht als Public-Daten auffassbar, sondern eher als Begleitdaten des Aufrufs (also Argumente) entweder der Haupt-step-Routinen, einer Initialisierung oder dergleichen. Oft werden den Abtastzeiten der FBlocks diese festen Routinen zugeordnet.

Vierte Frage: **Wie ist das mit Vererbung und Abstraktion?** Antwort: ***Kann man klären***. Diese Denkweise ist zwar meist nicht mit FBlock-Diagrammen verbunden, jedoch ist beispielsweise für FBlock-Diagramme der IEC61131 (Automatisierungsprogrammierung) definitiv die Vererbung in 2013 eingeführt worden. Sie ist zweckführend anwendbar, ist aber bei den entsprechenden Anwendern wenig angenommen. Vererbung und Abstraktion ist mit FBlock-Diagrammen möglich und sinnvoll.

Letzte Frage: Wie ist es mit **Composition, Aggregation, Association?** Eine Verwendung eines FBlocks innerhalb eines Modells, das nach außen wieder einen FBlock darstellt, ist von sich heraus eine Composition. Aggregations und Associations kann man bauen, das ist problemlos möglich und auch realisiert, auch wenn es nach außen nicht so genannt wird.

Zusatzfrage: Wie ist es mit der **in der Objektorientierung eigentlich ebenfalls definierten Verbindung von Objekten mit Events?** Antwort: ***4diac mit IEC 61499 verwenden***, basiert darauf vollständig. Weitere Antwort: Individuallösungen sind vorhanden.

2. Beispiele zu FBlock-Diagrammen, bei denen die Aussagen zur Objektorientierung mindestens teilweise zutreffen

Simulink

FBD-Diagramme der IEC61131 mit Vererbung

CFC für Simatic-TDC und dessen Vorgänger STRUC für Simadyn

IEC61499 mit 4diac

weiteres, eigentlich alls FBlock-Diagrammtypen.

3. Objektorientierung in Grafischer Programmierung, eigene Erfahrungen

3.1 STRUC-G und Simadyn-D

Meine ersten konkreten Aktivitäten der Objektorientierung in der Grafischen Programmierung sind in Simadyn-D mit STRUC entstanden. Dieses mittlerweile im www kaum auffindbare Tool ist bis ca. 2000 aktuell in Walzwerken etc. eingesetzt worden, auch der Transrapid in Shanghai ist in STRUC programmiert. Der Nachfolger ist https://de.wikipedia.org/wiki/Continuous_Function_Chart. STRUC-G benötigte noch einen Unix-Rechner zum Programmieren (kein Linux!). Die Herangehensweisen in STRUC und CFC sind in etwa gleich.

Die interne Funktionalität in den FBlocks in STRUC-G und CFC wird in C programmiert. Das Interface, die „Bausteinmaske“ ist ein Beschreibungsfile im speziellen Format. Die Grafischer Programmierung erfolgt durch Anordnen von Funktionsblöcken in Funktionsplänen, auf mehrere Seiten verteilbar, und Verbinden der Ein- und Ausgänge. Mit dem grafisch abgelegten Programm (das in STRUC-L auch als Listenform parallel vorliegt) und den Bausteinmasken ist ein C-Programm generiert worden, dass die Grafische Programmierung je eines Funktionsplanes (mehrere Seiten) widerspiegelt. Dieses generierte Programm ruft den Bausteinmasken entsprechend die inneren C-Funktionen auf.

Diese Herangehensweise entspricht weitgehend der in Simulink üblichen Programmierung (von Details abgesehen), wenn man S-Funktionen mit Kernen in C voraussetzt bzw. die Standard-FBlocks nutzt, die letztlich auch in Simulink mit C-Operationen abgebildet werden. Anders als bei Simulink wurden die Abtastzeiten und Abarbeitungsreihenfolgen bei STRUC-G manuell vorgegeben, Simulink erkennt diese automatisch aufgrund des Datenflusses.

Diese Herangehensweise entsprechend an sich auch der von 4-diac, wenn man damit nur ein Device betrachtet. Anstatt der Events gibt es die Abarbeitungsreihenfolge in der Abtastzeit. 4diac über mehrere Devices wird keinesfalls in STRUC abgebildet.

3.2 Datenverbindungen in STRUC-G

Da die Programmierung intern frei in C möglich war, konnte man interne Daten in den FBlocks definieren, über normale `typedef struct`, allokiert zur startup. Nun lag es sehr einfach auf der Hand, Datenverbindungen über sogenannte V4-Konnektoren (Datentyp uint32) weiterzugeben. Der Konnektor führte die Datenadresse von einem FBlock zu einem anderen, über die

Datenadresse konnte ein FBlock in C mit den Daten des anderen FBlock arbeiten. Das war für spezielle FBlocks durchaus eine gängige Herangehensweise.

Wenn diese V4-Datenzeigerverbindung initial ausgegeben wurde und danach nicht mehr verändert, dann entspricht dies einer Aggregation zwischen diesen FBlocks (Objekten der FBlock-class in der FBlock-Library). Ändert sich der Wert des Datenzeigers, darf auch 0 sein, dann ist dies eine Association. Beides ist zweckdienlich.

3.3 Nutzung der gleichen Herangehensweise in Simulink

In Simulink ist es für schnelle Abarbeitung interessant, die Kernfunktionen in C zu realisieren. Der Coder generiert dann mit dem Modell-Code optimale Aufrufe, so dass das Ganze in µs abarbeitbar ist. Teils wurden mit Modellelementen gebaute Reglerfunktionalitäten verglichen mit passender C-Programmierung, mit Effektivitätszuwachs halbe Rechenzeit für die C-Aufrufe, da die C-Funktionen in Zeilenprogrammierung optimal gedacht sind.

Problem ist die Zuordnung von Rechenschritten zu Abtastzeiten. Dies gelingt in C sehr leicht, wenn auf aufbereitete Daten über eine referenzierte `struct` zugegriffen werden kann. In diesem Fall ist es einfach klärbar, dass S-Function X die Daten in Abtastzeit Tn ggf. langsamer aufbereitet (Reglerparameter), die dann in der schnellen Abtastzeit Ts einfach abgegriffen werden, z.Bsp. für die schnelle Regelung selbst. Wenn diese Dinge direkt in Simulink modelliert werden, braucht es einige sogenannte „Rate Transitions“ im Modell, und ein Stoßgebet, dass man keine Rate Transition vergessen hat und daher der Codegenerator die Abtastzeitzuordnungen nicht richtig realisiert hat. Man merkt dies gegebenenfalls nicht sofort, nur wenn man genauer auf C-generierte Codes und Abarbeitungszeiten schaut. Das ist wesentlich komplexer als die einfach zu zeichnende Aggregation zwischen FBlocks, mit der die Abtastzeitzuordnung geklärt ist aufgrund der viel einfacheren Abtastzeitzuordnung für jeden FBlock selbst, als über den Signalfluss.

Diese Herangehensweise wurde auf einem Vortrag auf dem ESE-Kongress 2017 vorgestellt, nachlesbar im Artikel embedded-software-engineer/Elektronik Praxis: - Grafische objektorientierte Programmierung mit Simulink (Schorrig/Gerstl).

Mittlerweile wurde auch die Datenzuordnung von grafischen Modellteilen (nicht nur S-Funktionen) geklärt, mit spezifischen Datenhaltungs-S-Funktionen, die aber von außen parametrierbar sind und so keine C-Programmierung erfordern. Damit ist auch ein Funktionsblock, hinter dem ein grafisches Modell steht, objektorientiert auffassbar.

4. Situation in 4diac / IEC61449

4diac als Grafisches Tool für die Automatisierungsgeräteprogrammierung nach IEC 61499 benutzt ähnliche FBlock-Verbindungen wie die anderen bekannten Tools (hier nur STRUC, CFC, Simulink als Beispiele genannt). Besonderheit in 4diac ist die Orientierung auf **Eventverbindungen**:

4.1 Eventverbindungen

Mit dieser Grundidee ist Datenfluss und Control (Abarbeitungsfolge) vereint. In STRUC wurde die Abarbeitungsfolge mit einer Reihenfolgenummer und einer Abtastzeitzuordnung angegeben, der Datenfluss waren eigentlich nur Datenzugriffspunkte, unabhängig vom „Fluss“ der Daten. In Simulink ist dagegen der Datenfluss als einzig Bestimmendes definiert, die Abarbeitungsreihenfolge ergibt sich aufgrund des Datenflusses. Das ist konsequent für ein Denkmodell, allerdings nicht immer gut durchschaubar. Gibt es beispielsweise Schleifen (Rückkopplung im Datenfluss), folgt eine Fehlermeldung. Den Fehler kann man schnell beseitigen, indem an der richtigen Stelle eine Abtastverzögerung (ein 1/z-Glied) eingebaut wird. Wird schnell beseitigt das 1/z an der falschen Stelle eingebaut, gibt es eine unerwartete Totzeit, ähnlich wie es zu unerwarteten Totzeiten bei unpassend vorgegebenen Abtastreihenfolgen im STRUC kommt. Man kann hier sagen: Gleiches Problem, verschiedene Lösungen.

Insoweit ist die 4diac mit IEC 61499 besser, denn die Abarbeitungsreihenfolge wird grafisch gut sichtbar manuell angegeben. Laufen Eventverbindungen und Datenverbindungen offen sichtlich gut parallel, dann harmonieren diese. Die Daten sind den Events an FBlock Ein- und Ausgängen zugeordnet. Es werden also neue Daten immer zusammen mit dem Event bereitgestellt. Unabhängig vom Event können Daten „abgegriffen“ werden, also Daten, die nicht direkt mit dem Event der FBlock-Verbindung verbunden sind. Das entspricht den „*Daten aus einer anderen Abtastzeit*“ bei STRUC und Simulink.

Die Eventverbindungen in IEC 61499 haben noch eine andere grundlegende Bedeutung: Sie funktionieren auch über Gerätegrenzen hinweg. Man kann in einfacher Weise FBlocks bestimmen, die einzelnen Geräten zugeordnet werden. Bei Gerätegrenzen zwischen FBlocks werden dann Kommunikations-Daten-Strukturen automatisch angelegt, die Daten werden mit den Events in Telegrammen zwischen den Geräten transportiert. Das erspart eine Menge Arbeit, die etwa bei IEC61131 oder auch Simulink, STRUC mit entsprechender Kommunikationsplanung und manuellen Kommunikations-FBlocks erledigt werden muss.

Die Eventverbindungen entsprechen einer Objektorientierten Herangehensweise der Kommunikation zwischen den Objekten, wie sie beispielsweise in /Kai/ dargestellt ist.

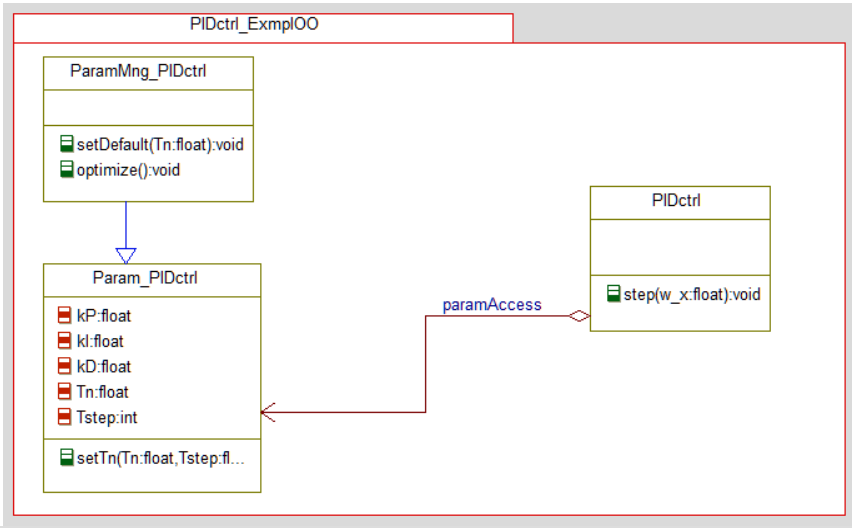
In 4diac /IEC 61499 ist es per se nicht möglich, mit einem Handle zu arbeiten und damit Daten in anderen FBlocks im gleichen Device zu referenzieren. Das geht zwar wenn man die Kerne der FBlocks in C/++ programmiert, wie die SFunction in Simulinj. Jedoch ist original StructureText an dieser Stelle vorgesehen. Im Unterschied zu IEC61131 werden die Datenverbindungen eben nur mit Event geführt, anders als in IEC 61131, siehe nächstes Hauptkapitel.

4.2 Eventkommunikation im ObjectModelDiagram

Also ist die Herangehensweise anders als in STRUC gefunden und Simulink geübt. Dazu ein interessantes Bild in Rhapsody (UML) gezeichnet:

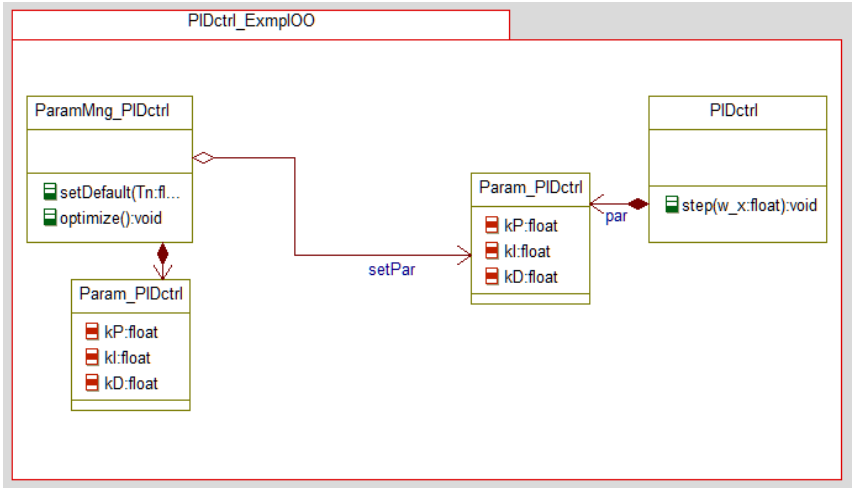
Assoziationen und Aggregationen auf Message-Daten

Das rechsstehende Bild zeigt die klassiche Lösung: Eine class PIDctrl braucht einen Parametersatz und findet diesen in `Param_PIDctrl`. Weil es schnelle Echtzeitverarbeitung sein soll, werden die Zugriffe direkt auf die nettoberechneten k-Faktoren per Referenz `par` ausgeführt. Die Implementierung ist also die einfache Gleichung eines PID-Reglers:




```
float wxp = par->Kp * w_x;
thiz->yi += par->kI * wxp;
thiz->y = thiz->yi + wxp + par->kD * (wxp - thiz->d);
thiz->d = thiz->wxp;
```

Diese Bild hat nur geringe aber bedeutsame Unterschiede:



* Anstatt der Vererbung auf den ParamMng enthält der ParamMng die Netto-Parameterwerte als Composition.

* Auch der PIDctrl hat eine Composition vom gleichen Typ.

* Über eine Aggregation `setPar` werden die Parameter offensichtlich kopiert.

Betrachtet man den Aspekt der Message-Kommunikation in der Objektorientierung, wie sie Alan Kay geäußert hat, dann können Assoziationen und Aggregationen auch als Ziel einer Messagekommunikation interpretiert werden. Die Message kann dargestellt werden der Aufruf einer Operation, aber ohne Erwartung eines Rückflusses von Daten (ohne return value, oder Rückschreiben über Argument-Referenzen). Das entspricht den üblichen Darstellungen in Sequenzdiagrammen der UML. Dort ist das *return* ein eigener Pfeil. Im obigen Bild ist die Aggregation `setPar` also eigentlich eine Message-Verbindung. Die Message selbst (die gerufene Operation) ist nicht dargestellt, ergibt sich aber direkt aus dem Kontext, etwa so:

```
void setPar(float kP, float kI, float kD);
```

oder

```
void setPar(Param_PIDctrl const* parData);
```

Man kann hierbei noch einen weiteren Schritt gehen. Anstatt dem Aufruf der obigen Operation, die die Parameterwerte übergibt, werden die entsprechenden Daten lediglich als Payload einer allgemeinen Message verstanden. Die Message wird vom Erzeuger (class `ParamMng_PIDctrl`) verpackt unter Kenntnis der Datenstruktur in 12 Byte. Der Empfänger entpackt und weiß aufgrund Informationen im Head der Message, dass es sich um Parameterdaten handelt. In konsistenter Software werden die 12 Byte also wieder entpackt in die gleiche Datenstruktur. In der Übertragungsstrecke ist die Kenntnis der Datenstruktur nicht notwendig, außer wenn die Message selbst beobachtet werden soll (WireSharc oder dergleichen). Dazu gibt es aber probate Hilfsmittel.

5. Objektorientierung in IEC61131

Die IEC 61131 ist der klassische Standard der Automatisierungstechnik und mit FunctionBlockDiagrammen (FBD) ebenfalls vertreten. Folgend ein Ausschnitt aus einem anderen Artikel, nicht veröffentlicht, in dem es um die Anwendbarkeit eines Plugin-Pattern geht:

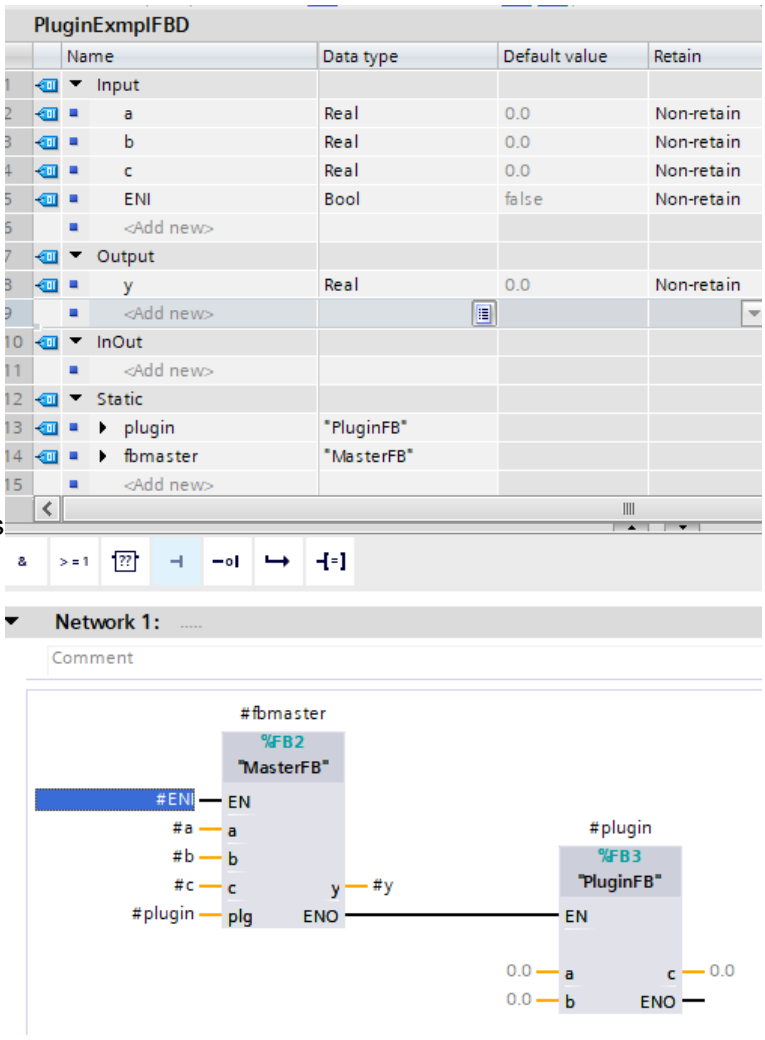
Das Problem ist, dass FBlock-Diagramme in der Automatisierungstechnik keinen Referenzmechanismus haben?

Es scheint, dass eine Lösung mit IO-Daten wie eine Referenz wirkt. Folgendes FBD ist mit dem TIA-Portal erstellt:

Der PluginFB enthält die Plugin-Funktionalität und hat dazu normale Input- und Output-Schnittstellen.

Der MasterFB oder Core-FB hat den Typ des PluginFB als InOut.

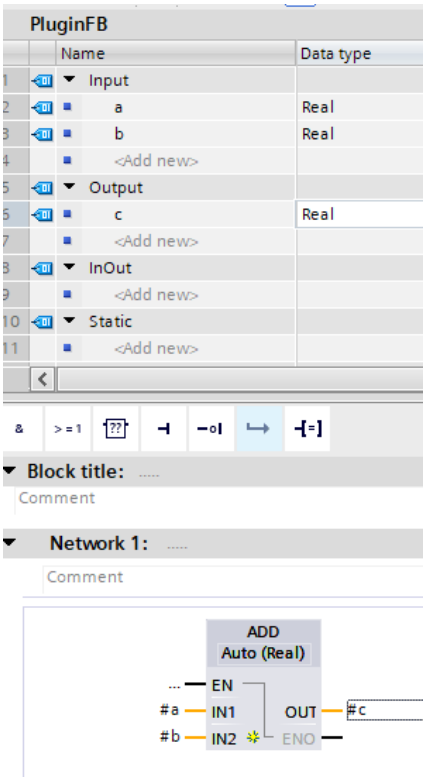
Wenn der Master-FB abgearbeitet wird, belegt dieser die Werte für a, b der plugin-Schnittstelle, die Werte werden in die Instanzdaten des Plugin-FB direkt geschrieben. Zur eigenen Verarbeitung holt er sich vom DB-Teil des PluginFB (aus dessen Instanzdaten) den Wert für c, der verarbeitet wird. Dies ist allerdings der Wert aus dem vorigen Verarbeitungsschritt (Abtastzeit bei zyklischer Bearbeitung). Es entsteht damit zwar eine Totzeit, anders als bei Objektorientierter Programmierung, in der die Verarbeitung des plugin-FB direkt im MasterFB gerufen werden könnte. Von dieser Totzeit soll einmal abgesehen werden.



Der Aufruf des Plugin erfolgt im FB-Diagram nach dem Master-FB.

Die Zusammenstellung ist hier aber noch übersichtlich. Es ist deutlich sichtbar, dass am plg-Connector der als Instanzdaten definierte PluginFB datenmäßig angeschlossen ist.

Der PluginFB sieht im einfachsten Fall wie rechtsstehend aus, die Plugin-Funktinalität ist also ein ADD. Das kann aber beliebig komplex sein oder auch in ST/SCL realisiert. Selbstverständlich können hier weitere Plugs existieren, die dann im aufrufenden FB-Diagram dargestellt werden müssen.



Hier nicht betrachtet sind die Möglichkeiten der Objektorientierten Programmierung, die die IEC-61131-3 in der Version von 2013 bringt. Möglicherweise kann man dort über ein Interface den Plugin-FB anschließen und innen wie einen innen vorhandenen FB aufrufen, das wäre ideal. (TODO).

... also hier wirken die IO-Daten wie eine Referenz auf einen anderen FBlock, dessen gesamte Interndaten.

Es geht zwar immer um die direkten Datenzugriffe, keine private-Kapselung, aber bei einer Umsetzung in generierten Code lassen sich an diesen Stellen auch passend getter und setter unterbringen. Letzlich sollte der Datenzugriff kein Thema sein.

6. Die gesamten FBlock-Beispiele zeigen, dass die FBlock-Programmierung im Kern objektorientiert ist.

Sie zeigen auch, dass die FBlock-Konzepte trotz Unterschiede sich ähneln, teilweise sogar ineinander überführbar sind. Es sind jedoch immer herstellerspezifische Konzepte, anders wie bei der UML, bei denen das „Unified“ im Namen steht. Die einzelnen Hersteller haben ganz andere Dinge im Fokus, als sich gegeneinander abzustimmen. Eher möchte man Unterschiede und Vorteile herausarbeiten. Ein „Unified“ kann nur von Kunden gefordert und akademisch gefördert werden.

6.1. Kann man aus der UML beitragen?

- man sollte. Die FBlock-Tools haben den Vorteil, dass deren Codegenerierung immer gelebt wird. Es geht nicht ohne, denn es sind vorderhand Programmierwerkzeugen, nicht Architektur- und Darstellungstools. Das haben sie gegenüber der UML mit der eher stagnierenden Nutzung von Codegenerierung im voraus.

Aber, die gesamten Mechanismen der UML, Backtracking von Requirements etc., wären für die FBlock-Tools anwendbar. Möglicherweise sogar recht einfach, wenn man eine Konvertierung in und aus einem Repository-Baum der UML findet. Die FBlock-Tool-Daten sind jedenfalls zugänglich.

Ein anderes Prinzip täte den FBlock-Tools gut: Die partielle oder mehrfache Darstellung von Modellteilen in verschiedenen Diagrammen. Bei den FBlock-Tools ist jeder aufgeführte FBlock ein Objekt. Zweimal in verschiedenen Diagrammen aufgeführt sind zwei Objekte. Nun kann man auch dort an der Codegenerierung arbeiten und mindestens im akademischen Bereich aufzeigen, dass die Instanziierung auch an eine Namensvergabe gebunden sein kann. Gleicher Name – gleiches Objekt, auch wenn es in verschiedenen Diagrammen auftaucht. Damit könnte man wie in den UML-ObjectModelDiagrammen jeweils interessierende Teile darstellen und so von einem reinen Programmierwerkzeug zu einem Darstellungswerkzeug kommen, ohne die Programmierfunktionalität zu schädigen.

Literatur

/Kay/ Alan Kay:The Early History of Smalltalk In: The second ACM SIGPLAN conference on History of programming languages. ACM. S. 78. 1. März 1993.

/Ho-etz/ www.etz.de/files/e21124zsh_hofer.pdf :Johannes Hofer: "*Objektorientiert programmieren mit dem TIA Portal* " in etz Heft 11/2012 (Zeitschrift Elektrotechnik und Automation von Herausgeber: VDE Verband der Elektrotechnik Elektronik Informationstechnik e. V.)

/TIA-OO/: Hofer, J.: SCL und OOP mit dem TIA Portal V11 – Ein Leitfaden für eine objektorientierte Arbeitsweise. Berlin · Offenbach: VDE VERLAG, 2012 (ISBN 978-3-8007-3436-8)

/61131/: INTERNATIONAL STANDARD IEC 61131-3 Second edition 2003-01, International Electrotechnical Commission, 3, rue de Varembé, PO Box 131, CH-1211 Geneva 20, Switzerland,Telephone: +41 22 919 02 11 Telefax: +41 22 919 03 00 E-mail: inmail@iec.ch Web: www.iec.ch

/61331OO/ 65B/858/FDIS FINAL DRAFT INTERNATIONAL STANDARD
Project number IEC 61131-3 Ed 3, Ausgabedatum 2013-01-18
Titel: IEC 61131-3 Ed 3: Programmable controllers - Part 3: Programming languages

/61499/: 65B/845/FDIS FINAL DRAFT INTERNATIONAL STANDARD,
Project number IEC 61499-1/Ed.2, Ausgabedatum 2012-10-19
Title: IEC 61499-1/Ed.2: Function blocks - Part 1: Architecture

/61499-OO/ IEC61499 und OO, in computer-automation.de: Host Meyer: "*Steuerungs-Engineering: Die Migration von der IEC 61131 zur IEC 61499, Objektorientierung oder: Was ist ein Objekt?"* in computer-automation.de vom 2013-06-11