Dr. Hartmut Schorrig, www.vishia.org

Objektorientierte, Datenfluss, Funktionale Programmierung und Messagekopplung / Eventverarbeitung
Methodik und Vergleich, Anwendung in Java, C/++ und in der Automatisierungsgeräteprogrammierung

Abstrakt

Es wird heute mehr von Funktionaler Programmierung geredet. Einige Programmiersprachen sind dafür ausgelegt, andere Programmiersprachen unterstützen dies, so auch Java seit der Version 8 mit den Lambda-Ausdrücken. So ist es zu lesen. Doch, was ist das Grundkonzept der Funktionalen Programmierung, wie kann man sie anwenden, wie stellt sie sich gegenüber den anderen Paradigmen auf ?

Funktionale Programmierung ist keine Leistungseigenschaft einer Programmiersprache sondern eine Herangehensweise. Richtig angewendet im Mix mit den anderen Herangehensweisen bietet sie Vorteile.

Inhaltsverzeichnis

1. Geschichte: Spaghetti-, Strukturierte Programmierung und Objektorientierung

Am Anfang der Programmierung standen Maschinenbefehle. Die ersten Programmiersprachen lieferten eine den Maschinenbefehlen folgende lesbare Formulierung. Damit war das "`GOTO`" als pendant zum Sprungbefehl `jmp` war damit natürlicher Bestandteil der Programmiersprachen. '`goto`' ist in der Programmiersprache C und C++ immer noch präsent und verwendbar, es verwendet zum Glück aber fast niemand.

Wenn auf Assemblerniveau programmiert werden muss, dann können Sprünge so angelegt werden, dass sie eine strukturierte Programmstruktur ergeben. GOTO ist ebenfalls in Windows-Batchfiles noch üblich. Auch hier empfiehlt es sich GOTO nur sehr strukturiert anzuwenden.

Es war in den 60-ger Jahren mit Algol als Vorreiter eine entscheidende Entwicklung, dem Programmierer Mittel in die Hand zu geben, damit nicht mehr manuell auf die strukturierte Verwendung von goto selbständig geachtet werden muss. Die sogenannte Strukturierte Programmierung war über lange Zeit ein Diskussionsthema (ob unbedingt, oder in Sonderfällen doch goto). Eine Grafische Ausprägung erfuhr sie mit den Nassi-Shneidermann-Diagrammen (link Wikipedia) oder auch Struktogrammen. Mit dem "Struktogrammeditor" konnte man die einzelne innere Strukturen aufklappen, die umgebende Struktur verbergen, und so beliebig tief schachteln. Die Programme waren dann im Quelltext eigentlich nicht mehr lesbar. In diesem Sinn ist die Anwendung der Objektorientierten Programmierung mit der Tendenz zu eher kleinen Operationen ein Segen für den Durchblick durch die Software.

Die Objektorientierte Programmierung hat ihre Vorläufer im universitären Bereich auch bereits in den 60-ger Jahren, ähnlich wie auch die Funktionale Programmierung. Ein Programmierparadigma ist aber erst dann präsent (gegenwärtig) wenn es allgemein verwendet wird. So kann die Anfangszeit der Objektorientierten Programmierung mit C++ auf die 90-ger Jahre gelegt werden.

2. Objektorientierte Programmierung

Im Wikipedia-Eintrag für Objektorientierte Programmierung - Wikipedia wird einleitend ein Zitat von Alan Kay, Mitentwickler von Smaltalk und einer der Gründungsväter der Objektorientieren Programmierung, präsentiert:

1. Everything is an object,

2.Objects communicate by sending and receiving messages (in terms of objects),

3.Objects have their own memory (in terms of objects),

4.Every object is an instance of a class (which must be an object),

5.The class holds the shared behavior for its instances (in the form of objects in a program list),

6.To eval a program list, control is passed to the first object and the remainder is treated as its message

– Alan Kay: The Early History of Smalltalk (1993) /Kay/

1. Alles ist ein Objekt,

2.Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen),

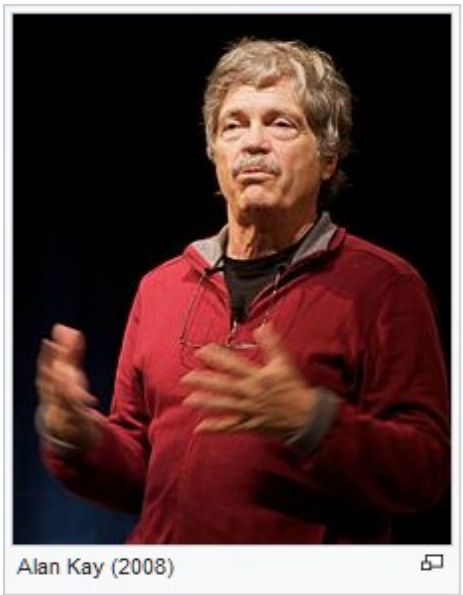
3.Objekte haben ihren eigenen Speicher (strukturiert als Objekte),

4.Jedes Objekt ist die Instanz einer Klasse (welche ein Objekt sein muss),

5.Die Klasse beinhaltet das Verhalten aller ihrer Instanzen (in der Form von Objekten in einer Programmliste),

6.Um eine Programmliste auszuführen, wird die Ausführungskontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt

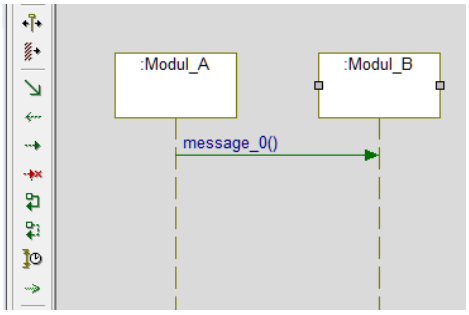
(Übersetzung laut Wikepedia, abgerufen in o.g. Link am 2019-07-15)


Alan Kay (2008)

Alan Kay, der sehr viel in Programmiersprachen, Paradigmen und deren praktische Anwendung unterwegs war, dabei nicht nur technisch orientiert (Jazzgitarrist, Theater, Klassiche Orgel), hat den Aspekt der Eventkommunikation, wie sie bekannt ist in einigen Anwendungen (Statemachines, Reaktive Programmierung) unmittelbar mit der Objektorientierten Programmierung verbunden. Das rechtsstehende Bild ist in Screenshot aus dem oben genanntem Wikipedia-Artikel, Rechte: *Wikimedia Commons, the free media repository*

In der praktisch angewandten Objektorientierung spielt aber das Messaging nicht diese Rolle. In den gängigen Objektorientierten Programmiersprachen (C++, Java, C#) gibt es für Messaging und Eventhandling zwar spezielle Libraries, aber keine Einbindung direkt in die Syntax und Basisanwendung der Programmierung. Gewöhnlich wird Objektorientierung in erster Linie mit den Begriffen Abstraktion,

Ableitung (Derivation), Vererbung (Inheritance), Assoziation, Aggregation und Composition aus UML-ObjectModelDiagrams verbunden, und nur für Spezialfälle die Message / Eventkommunkation unabhängig von der Objektorientierung benutzt.



Neben den Statecharts in der UML mit ihren Eventhandling-Mechanismen fällt aber in UML-Sequenzdiagrammen auf, dass der Aufruf einer Operation als "m*essage*" bezeichnet wird, obwohl im Designmode damit direkt eine Methode der Klasse angelegt oder angewählt wird. Das rechtsstehende Bild ist ein Screenshot aus Rhapsody in der Version 7.3 (etwa aus 2007), auch in einer aktuelleren UML-Version, am Beispiel Enterprise Architect sieht dies ähnlich aus (getestet in Version 12.1).

In der bekannten C++-Programmierumgebung QT (https://www.qt.io/) gibt es seit Anfang in den 90-gern und aktuell die ***Signal-Slot-Technologie***. Hierbei wird in einfach anwendbarer Weise ein Mechanismus bereitgestellt, der sogenannte Signale an andere Instanzen senden kann. Dies ist eine Message-Technologie, nur anders bezeichnet. QT hat also das Messaging von Anfang an mit seinem Objektorientierten C++-Konzept verbunden. QT ist zwar weit verbreitet, aber auch nur eine wenn auch größere Insel (angelehnt an den Buchtitel "*Java ist auch nur eine Insel*" von Christian Ullenboom:
http://openbook.rheinwerk-verlag.de/javainsel/

Es fehlt also eine übergreifende Sicht auf dieses Thema.

Man kann die Objektorientierung in drei Ebenen einteilen:

1) Grundlegende Definition der Objektorientierung: **Die Objektorientierung bündelt Daten, bildet damit Datenobjekte, und verbindet diese mit den zugehörigen Bearbeitungs- und Zugriffsoperationen. Die Beschreibung der Objekte und der Operationen (Typbeschreibung) werden in einer Klasse gebündelt**
2) Abstraktion, Vererbung, Klassenbeziehungen: **In der Objektorientierung kann der Zugriff aus einer auf eine andere Instanz einer Klasse über eine abstrakte Sicht erfolgen. Eine Klasse kann mit Abstraktionsebenen versehen werden. Auf einer Abstraktionsebene basierend kann es abgeleitet spezifische Ableitungen geben. Dabei werden über den abstrahierten Zugriff auch Operationen (Methoden) des abgeleiteten Klassentyps direkt aufgerufen, bezeichnet als dynamisches (spätes) Binden.** Das ist das Thema mit dem sich Objektorientierung gemeinhin in den Objektorientierten Programmiersprachen beschäftigt.
3) Messagekommunikation: **Die Kommunikation zwischen den Instanzen von Klassen (Objekten) kann über einen Message-Mechanismus erfolgen. Damit sind die Objekte stärker getrennt als etwa mit Assoziationen und Aggregationen. Eine Message löst den Aufruf einer Operation (Methode) in der Zielklasse aus und kann Daten enthalten.**

Folgend werden diese Eigenschaften der Objektorientierung an Programmiersprachen und Programmierkonzepten bewertet.

2.1 Zugriff aus einer Instanz auf Daten und Operationen einer Instanz einer anderen Klasse

1. **Der Zugriff** von den Operationen auf die Daten erfolgt in der Regel (nicht zwangslläufig) **über Referenzen**. Referenzen sind maschinentechnisch eine Speicheradresse, die aber compilertechnisch mit dem entsprechenden Typ qualifiziert ist.

2. **Abstraktion**: Es kann ein Basistyp definiert werden, der davon abgeleitete Objekte repräsentieren kann. Mithilfe der Abstraktion ist eine Verallgemeinerung der Nutzung eines Objektes definierbar, für verschiedene Objekttypdefinitionen werden gemeinsame Eigenschaften definiert. Es ist nun möglich, auch über eine Referenz des Basistyps auf das jeweilige Objekt zuzugreifen.

3. Es wird unterstützt, dass auch **bei Zugriff über eine Basistyp-Referenz** bei bestimmten Operationen **die Operationen des tatsächlich vorhandenen (abgeleiteten) Typs aufgerufen wird**, ohne dass der abgeleitete Typ im Aufrufkontext bekannt ist.

Es wird hier der Begriff **Operation** verwendet, der identisch mit **Methode** ist.

Der Punkt 3 beschreibt das späte dynamische Binden als sinnvolle und gängige Eigenschaft der Objektorientierung. Die drei Punkte sind aber tatsächlich sekundäre Eigenschaften. Auch wenn das dynamische Binden oder die Abstraktion nicht benutzt wird, weil nicht notwendig, oder auch in einem bestimmten Kontext nicht möglich, dann gilt die primäre Definition der Objektorientierung dennoch.

Direkte Datenzugriffe, private und public

Zunächst zur private-Kapselung. Dieser wird häufig eine hohe Bedeutung beigemessen, und dann doch unterwandert. Was ist gemeint:

Die private-Kapselung ist zunächst ein Zuständigkeit - Übereinkommen zwischen Programmierern. Die private-gekennzeichneten Daten einer Klasse können innerhalb des Quellcodes dieser Klasse beliebig geändert werden, ohne dass eine Abstimmung mit nutzenden Softwareteilen erfolgen muss. Es muss lediglich gesichert sein, dass die public Operations unverändert bleiben. In der nicht-Objektorientierten C-Programmierung kann eine Änderung von Daten schonmal viele Programmteile betreffen, die dann nachcompiliert oder sogar nachgebessert werden müssen.

Wenn es automatisch generierte getter und setter-Routinen für alle private Daten gibt oder wie in C# sogar die einfache Nutzung der private-Variablen zwar über synthetische get- und set-Routinen geht, man braucht diese nicht mal mehr explizit zu deklarieren, dann wird dieses Prinzip vollständig unterwandert.

Abgesehen von dem Getter/Setter-Thema ist es nun üblich oder empfohlen, nicht direkt mit den Variablen einer anderen Klasse zu arbeiten, aber es hält sich kaum jemand immer daran. Warum, weil es schneller zum Ziel führt, direkt auf die Variable zuzugreifen ("*Zeit ist Geld, Time to market*").

Erfolgt der Zugriff nur lesend, dann kann man in Java beispielsweise eine Variable als `final public` kennzeichnen, nur im Constructor belegen und allen den Inhalt zur Verfügung zu stellen. Diese Arbeitsweise ist eigentlich in Ordnung, wenn diese entsprechenden readonly-Daten als public-Eigenschaften, also als nach außen bekannte Schnittstelle definiert und so verstanden werden.

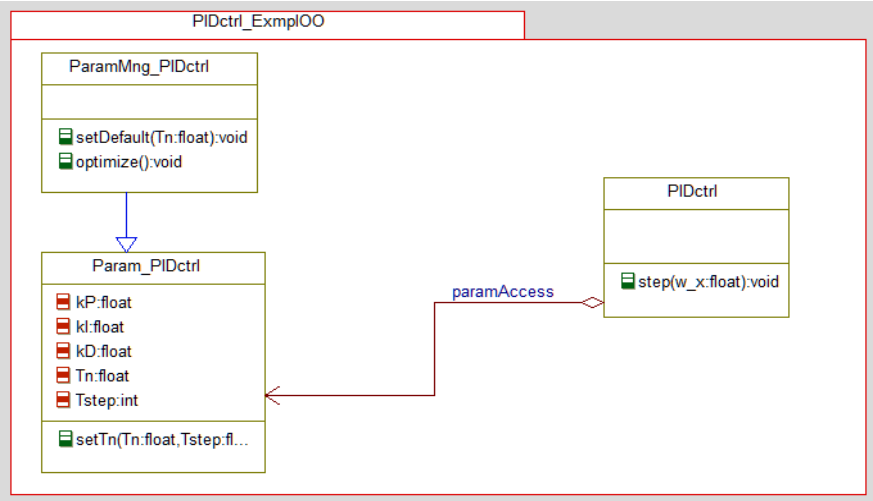
Wenn jedoch eine Klasse in einer anderen Klasse Variable direkt manipuliert, oder fast direkt über den Setter manupuliert, dann ist man fast beinahe wieder bei dem Thema "*alle arbeiten mit globalen Variablen*" und sollte die Frage stellen "*ist das noch Objektorientierung?*". Auch eine indirekte Manipulation durch bestimmte Operationen ist zwar regelkonform zu OO und etwaigen Styleguides, aber nicht besser.

Sind Assoziationen und Aggregationen der UML Referenzen?

Nach dem allgemeinen Verständnis: "*selbstverständlich ja*". Die Frage, ob man damit direkt die Daten der anderen Klasse manipuliert, ist oben erwähnt als Frage der private-Kapselung. Unabhängig davon, Operationen (Methoden) werden gerufen mit der Referenz auf die jeweilige andere Klassen-Instanz. Beim Aufruf einer public-Operation gibt es also keinen Unterschied ob diese aus einer anderen Operation der eigenen Klasse heraus gerufen wird oder aus einer anderen Klasse. Das wird nirgends kontrolliert und nicht hinterfragt.

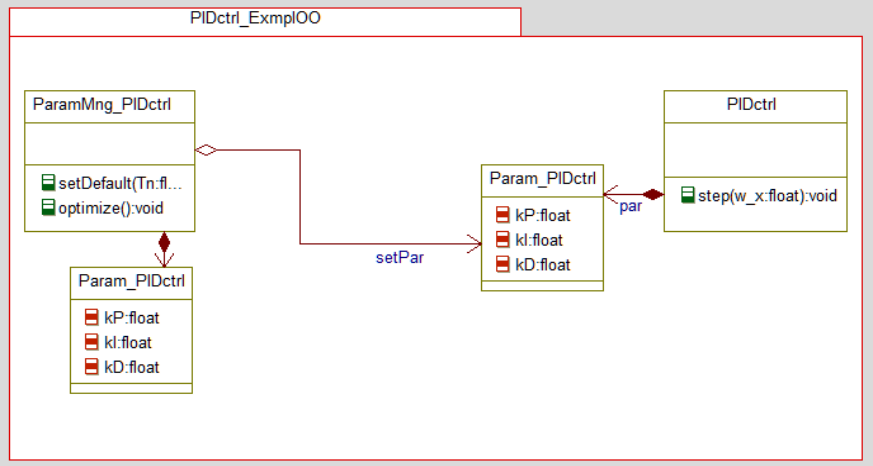
Es sind damit beliebig Seiteneffekte erzielbar, egal ob erwünscht oder fehlerhaft, egal ob vereinfacht über public-Variablenzugriff oder über spezielle Operationen, also formell clean.

2.2 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.

Diese Herangehensweise hat noch eine zweite wesentliche Bedeutung für die aufkommenden Multicore-CPUs auch im Bereich des Embedded Control:

* Die Parameteraufbereitung (ParamMngPIDctrl) läuft sehr wahrscheinlich in einem anderen Thread weil seltener berechnet. Man denke beispielsweise an eine Paramterumschaltung entsprechend der Außenumgebung des zu regelnden, oder an eine Parameteroptimierung. Die Berechnung des PIDctrl erfolgt im schnellen Zeittakt, beispielsweise 10 µs (!), Parameter werden aller Millisekunde optimiert.

* Damit stehen die Parameterwerte über klassische Referenzierung in einem anderen Bereich des Speichers. Der Zugriff darauf bedeutet, die Werte zunächst in den Cache zu laden. Das wird aber leider in jedem 10 µs - Abtastschritt wiederholt, weil der Zugriff möglicherweise über Synchronisationsmechanismen und volatile-Kennzeichnung erfolgen sollte. Das ist unoptimal.

* Mit dem Message-Konzept werden dagegen die Daten aktiv dann gesendet, wenn sie tatsächlich geändert sind. Das ist viel seltener. Die Konsistenz beim Senden ist gewahrt. Auf der Empfängerseite kann man wegen der Konsistenz mit einem Wechselpuffer arbeiten.

* Auf der Empfangsseite sind die Daten direkt mit denen des PIDctrl verwoben, es ist eine Composition. Im Zusammenhang mit dem Wechselpuffermechanismus werden sie nur dann neu vom Hardware-RAM in den CPU-internen Cache geladen, wenn die Pufferumschaltung erfolgt. Ansonsten stehen sie zwischen den Abtastschritten im Cache bereit. Der Zugriff in der 10 µs-Steptime ist schneller.

Damit erscheint das Message-Konzept, wie es von Alan Kay in 1993 publiziert wurde, in einem ganz aktuellem Licht.

Der Zugriff auf die eigenen (composite) Daten darf nun auch direkt erfolgen und nicht über ungeliebte Getter.

Ein Seiteneffekt ist verhindert. Es könnte ja der schlaue PIDctrl-Programmierer auf die Idee kommen, selbst die Parameter zu optimieren. Relevante Informationen sind im Regelalgorithmus abgreifbar. Bei direkter Referenzierung würde das kaum auffallen. Programmierer X hat also `kI` in bestimmten Fällen variiert, hat dann aber eine für ihn günstigere Arbeitsanstellung gefunden und wenig dokumentiert. Der Nachfolger wundert sich über unerklärtes Reglerverhalten, findet aber nicht den etwas undurchsichtig programmierten Zugriff auf `kI`. Das ist Programmieralltag. Mit dem Message-Konzept ist eine losere Kopplung diese eigentlich unabhängigen Module erreicht, die solche Konstellationen verhindert.

2.3 Kann man in C objektorientiert programmieren ?

Klare Antwort: Ja

* Die Datenbündelung und Datendefinition erfolgt mit dem Sprachelement `struct`.

* Die Zuordnung der Operationen zu den Daten erfolgt mit dem Typzeiger auf die zugehörige `struct`, der an bestimmter Stelle (empfohlen als erstes, Name `thiz`) der Aufrufargumente der C-Funktion übergeben wird. Zusätzlich soll die Strukturzugehörigkeit in der Namensgebung der C-Operationen verankert sein, empfohlen als Suffix.

* Die Abstraktion erfolgt über eine Basis-`struct` am Anfang der betreffenden struct-Typdefinition. Damit ist Einfachvererbung realisierbar (wie in Java).

* Die späte dynamische Bindung ist grundsätzlich über den Funktionsmechanismus der C-Funktionspointer möglich. Dazu muss in bestimmten Basisdaten jeder struct-Typdefinition die Referenz auf die Tabelle der instanztypbezogenen Funktionspointer gegeben sein. In C++ ist das auch nicht anders, dort befindet sich der Zeiger auf die sogenannte 'virtual table' ebenfalls am Anfang der Daten, wobei wegen der Mehrfachvererbung in C++ dieses Konzept dort etwas komplexer ist.

Man erhält somit in C eine Objektorientierung, die alle Belange mit Einfachvererbung unterstützt und einige Nachteile von C++ vermeidet.

Das Schreiben der Programme ist aufwändiger als in C++, der C++-Compiler für C++ würde einiges an manuellen Formulierungen abnehmen. Dafür ist die **Nähe zum Maschinencode wie sie für C bekannt ist, gewahrt**.

Die Lösung hat nichts mit der anfänglichen Adaption von C nach C++ aus den 90-gern zu tun sondern ist eine native Objektorientierte Lösung für C.

Beispiele dafür als Internet-Link:

* objektorientierte-programmierung-mit-c in www.embedded-software-engineering.de: Frank Listing (MicroConsult GmbH) / Franz Graser (Redakteur): Objektorientierte Programmierung mit C, veröffentlicht 2019-04-05.

* www.vishia.org/emc: Meine eigene Website zu dieser Thematik

* Eine Forumdiskussion über C objektorientiert, C++ und Linus Torvalds : https://www.c-plusplus.net/forum/topic/342279/objektorientiert-programmieren-in-c

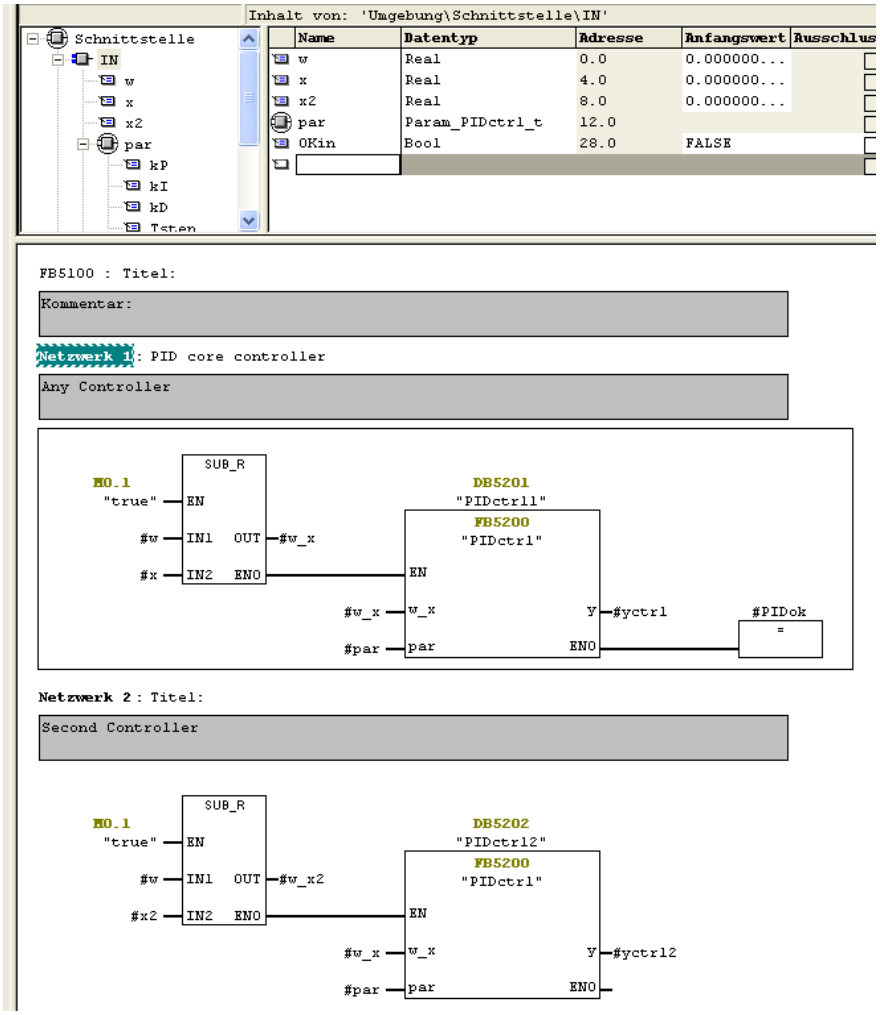
2.4 Objektorientierung in den Sprachen der Automatisierungstechnik nach IEC61131-3 und IEC 61499

Folgt man der Primären Definition in Kapitel 2.1 Zugriff aus einer Instanz auf Daten und Operationen einer Instanz einer anderen Klasse, Seite 6, dann ist jegliche Programmierung in FBD (FUP, Funktionsplan) mit FB und DB-Bausteinen objektorientiert. Denn:

* Die Operation steht in einem FB (`FUNCTION_BLOCK`). Dort werden Daten in SCL oder ST (Structure Text) zwar definiert aber nicht angelegt.

* Die Zuordnung eines DB (Data Block) zum FB im älteren Simatic-Manager ist zwar manuell auszuführen und damit etwas nervig, aber es ist genau die Instanziierung der Operation.

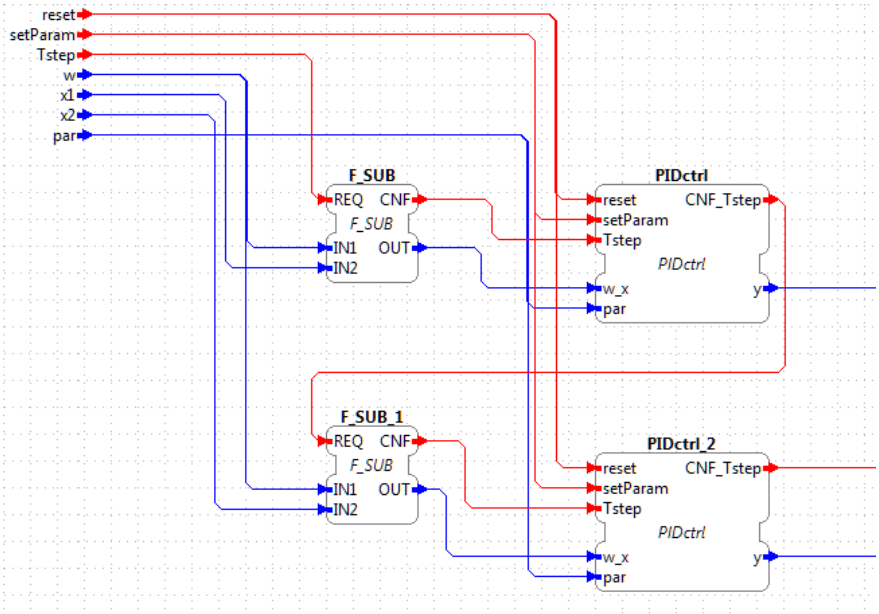
Das rechtsstehende Bild zeigt den PIDctrl mit Parameter der auch dem Kapitel 2.2 Assoziationen und Aggregationen auf Message-Daten, Seite 8 Pate stand, im etwas älteren Simatic-Manager (vor 2010):



Es sind zwei Instanzen des FB 5200 enthalten, die unterschiedliche DB 5201 und DB5202 nutzen. Die Inhalte der DB werden automatisch von den Variablendefinitionen in SCL in den FB bestimmt weil es "Instanzdatenbausteine" sind.

Interessant ist die Aggregation zu den Parametern. Diese ist aus dem Funktionsplan heraus auf den Input `par` gelegt. Dessen Typ ist ein UDT (User defined Datatype), symbolisch passend mit `Param_PIDctrl_t` bezeichnet. Der Typ steht in der Definitionstabelle oben, aber er lässt sich hier auch direkt aufklappen. Der Parameter-FB und DB selbst sind in einer anderen Abtastzeitschale letztlich in S7 über einen anderen OB (Organisationsbaustein) aufgerufen und werden beim Aufruf dieses Funktionsplanes jeweils kopiert. Das Kopieren erfolgt immer vor dem Aufruf, insoweit nicht optimal, aber im PIDctrl-Algorithmus (in SCL) stehen die Daten wie in Kapitel 2.2 Assoziationen und Aggregationen auf Message-Daten, Seite8 beschrieben als Composition (in den Eingangsdaten) bereit. Ein mehrfacher Zugriff ist also optimal schnell.

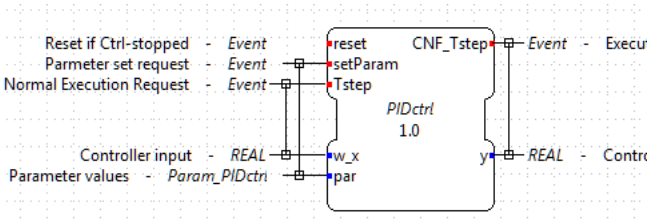
Im aktuellen TIA-Portal sieht diese Sache ähnlich aus. Die Bedienung ist erleichtert, Instanzdatenbausteine werden mit wenigen Klicks selbstverständlich angelegt, die Numerierung ist zwar noch vorhanden, aber nicht mehr auf User-Level zu beachten. Der Standard ICE 61131-3 verfügt aber über Erweiterungen aus dem Jahre 2012 für Objektorientierung mit Vererbung und Abstraktion /61131-OO/, die auch im TIA-Portal verfügt sind, beschrieben in www.etz.de/files/e21124zsh_hofer.pdf :Johannes Hofer: "Objektorientiert programmieren mit dem TIA Portal " in etz Heft 11/2012 /Ho-etz/, siehe auch /TIA-OO/, das entsprechende Fachbuch.



Das Problem des fortwährenden Kopierens der Daten wird in dem neueren Standard IEC /61499/ besser gelöst. Dort gibt es genau die Message- (oder Event-)-Kopplung der FunctionBlocks, wie sie Alan Kay für die Objektorientierung beschrieben hat. Man kann einen FB in IEC 61499 als Objekt auffassen.

Die Message-Linien sind hier rot, die Datenverbindungen sind blau. kommt das Input-Event `setParam`, das in der Message-Queue in der entsprechenden Ressource (Abarbeitungseinheit) gespeichert ist, dann werden die zugehörigen Daten in der Message gespeichert vom Typ Param_PIDctrl an die zugehörigen beiden PIDctrl-FBlocks übertragen und in diesen FBlocks intern gespeichert. Mit dem Abarbeitungsevent Tstep, das beispielsweise aller 10 µs zyklisch kommt, verwendet dann der PICctrl schnell seine intern gespeicherten Werte. Erst wenn der hier sich außen befindliche Parameterdienst wieder neue Werte hat, gibt er diese mit der EventKennzeichnung setParam in die MessageQueue. Nur dann wird Abarbeitungszeit für das Kopieren benötigt.

Die Zuordnung der Daten zu Event-Inputs und Outputs erfolgt in der hier auch grafischen Interfacedefinition, hier am PIDctrl gezeigt.



Siehe auch /61499-OO/ aus 2013. Die nächste reguläre Überarbeitung des IEC 61499-Standards wird in 2020 erfolgen.

Die bisher vorgestellten Objektorientieren Ansätze entsprechen der Ebene 1) in der Einleitung zu Kapitel 2 Objektorientierte Programmierung, Seite 4: "*Die Objektorientierung bündelt Daten, bildet damit Datenobjekte, und verbindet diese mit den zugehörigen Bearbeitungs- und Zugriffsoperationen. Die Beschreibung der Objekte und der Operationen",* allerdings mit einer wesentlichen Einschränkung: Pro Objekt ist nur eine Operation möglich.

Mit der Standardisierung der IEC 61131-3, 3. Ausgabe vom 2013-01-18 (/61331OO/) wurde nunmehr nicht nur mehrere Operationen (dort wieder *Methoden* genannt) pro CLASS; die FUNCTION_BLOCK ersetzt, zugelassen. Es wurde auch die wesentlichen Features der Ebene 2) laut der Einleitung der Objektorientierung Kapitel 2 im Standard aufgenommen, präsent mit den Keywords: EXTENDS ,

SUPER, OVERRIDE, FINAL mit den entsprechenden Mechanismen. Man hat somit in der Automatisierungstechnik nach IEC 61131-3 etwa gleiche Möglichkeiten wie in C++, Java, C# wobei diese Programmiersprachen im Standard als vergleichbar explizit erwähnt werden. In diesem Standard ist allerdings nicht der Aspekte 3) der Eventkommunikation enthalten. Dieser ist allerdings mit der IEC 61499 präsent. Es bleibt abzuwarten, ob in der Neuausgabe der IEC 61499, die für 2020 erwartet wird, nun beides vereint ist. Die IEC 61499 soll auf der IEC 61131-3 basieren, also auch in Zukunft auf der 3. Ausgabe, das bliebe abzuwarten.

Die Automatisierungsprogrammierung ist einige Jahrzehnte traditionell ohne die Objektorientierung ausgekommen, aber neuere Veröffentlichungen ab ca. 2010 legen nahe, OO stärker zu verwenden:

* Stefan Hennecken (aus 2010): IEC 61131-3: Vorteile der objektorientierten Spracherweiterungen
* www.sps-magazin.de/...nr=112569: Objektorientiert programmieren in der SPS
* Roland Wagner: Objektorientierte Programmierung von Automatisierungsapplikationen im Kontext der IEC61131-3, Vortrag auf dem ASE-Kongress 2019
  http://www.ase-kongress.de/de/programm

2.5 Objektorientierung in Java

Java wurde als neue Sprache geschöpft als nicht nur die Grundlagen der Objektorientierung mit Smalltalk & co bereits gelegt waren, sondern auch mit C++ eine über 5 Jahre währende Einsatzbreite der Objektorientierung in der Praxis vorhanden war.

Die Mehrfachvererbung aus C++ wurde in Java nicht übernommen. Java unterstützt bis heute erfolgreich nur Einfachvererbung. Es gab zwar auch in Java immer wieder Stimmen der Forderung nach Mehrfachvererbung. Der Gegentrend, etwas weniger Vererbung, anstelle dessen Aggregation ist besser für Softwarestrukturen, steht dem aber auch entgegen.

Java kennt ein Konzept der inneren nicht statischen Klassen, die implizit eine Referenz auf die äußere Klasse enthalten (explizit geschrieben als `TypOuterClass.this`). Die implizite Referenz lässt sich für eine `static class` auch explizit formulieren, was gegebenenfalls übersichtlicher ist. Wichtiger ist, dass privat-Datentypen der äußeren Klasse direkt angesprochen werden können. Über diesen Mechanismus ist eine Aggregation anstatt Vererbung sehr gut formulierbar. Wenn man die inner Klasse vererbt, hat man ähnliche Möglichkeiten wie bei der Mehrfachvererbung.

Die Überladbarkeit aller Operationen wurde als default-Verhalten definiert, anders wie in C++ oder C#. Bei letzteren muss `virtual` vor eine überladbare Operation geschrieben werden. Bei Java muss `final` davor geschrieben werden, wenn sie nicht überladbar ist. Das kommt zwar letzlich auf das Gleiche heraus, aber wer schreibt schon vor die meisten Operationen, die eigentlich nicht überladbar sein sollen, davor `final`. Damit kann der Compiler aber weniger gut optimieren. Die default-Überladbarkeit kann aus heutiger Sicht als überzogene Objektorientierung der Anfangszeit bewertet werden, ist aber als Spracheigenschaft nunmal festgeschrieben.

Dass die Grundtypen `char`, `short` usw. bis `double` keine Objekte sind, stört nur Theoretiker oder Schlauberger. Es ist von praktischen Nutzen für den Ablaufcode. Das Automatische Wrappen mit den entsprechenden Objekt-Typen `java.lang.Character`, `Short` usw. bis `Double` löst auf einfache Weise das Problem, wenn diese Typen als `Object` gebraucht werden, beispielsweise als Key in Maps.

Bei der Kapselung ist es nach Meinung des Verfassers etwas ungewöhnlich oder auch unglücklich, dass die `protected`-Kennzeichnung als "*zugriffsfähig in abgeleiteten Klassen*" gleichzeitig "*zugriffsfähig innerhalb aller Klassen des Package*" bedeutet (*package private*, ohne extra Kennzeichnung). Hat man große zusammenfassende Packages, dann sind viele Zugriffsmöglichkeiten damit nicht verriegelt.

In Java sind die Message- und Eventmechanismen nicht im Standardsprachumfang verankert, aber in speziellen Frameworks, häufig für die Serverseitige Java-Programmierung. Eventmechanismen kommen selbstverständlich in den Standard-Libraries der Grafikprogrammierung vor, als Callback für User-Actions.

3. Funktionale Programmierung

Ähnlich wie einleitend zur Objektorientierung bemerkt, gibt es viele Unklarheiten und Missverständnisse, wenn für die Funktionale Programmierung wie beispielsweise in Funktionale Progarmmierung (Wikipedia) (abgerufen am 2019-07-16) beschreiben ist:

Unter einer rein funktionalen Programmiersprache wäre eine Programmiersprache zu verstehen, die isomorph zum Lambda-Kalkül ist

Der Wikipedia-Artikel beschreibt unter anderem in der Einleitung die eigentlichen Eigenschaften:

Die besonderen Eigenschaften der funktionalen Programmierung ermöglichen es, auf die in der imperativen Programmierung benötigten inneren Zustände eines Berechnungsprozesses ebenso zu verzichten wie auf die zugehörigen Zustandsänderungen, die auch Seiteneffekte genannt werden. Der Verzicht darauf vereinfacht auf der einen Seite die semantische Analyse eines Computerprogramms erheblich und eröffnet auf der anderen Seite weitreichende Möglichkeiten zur regelbasierten, algebraischen Programmtransformation und -synthese. Daraus ergibt sich die erhebliche praktische Bedeutung der funktionalen Programmierung für die Informatik.

An diesem Satz ist allerdings der Hinweis auf die "*in der imperativen Programmierung benötigten inneren Zustände*" irreführend. Denn, die imperative Programmierung selbst benötigt für sich genommen keine Zustände. Gemeint ist, das imperativ häufig zweckmäßig mit gespeicherten Daten gearbeitet wird. Zudem sollte man Zustandsänderungen und schlecht überschaubare Seiteneffekte als zwei verschiedene Dinge behandeln.

Daher folgende einfache Definition:

Primäre Definition: **Funktionale Programmierung liegt dann vor, wenn eine Routine ihr Ergebnis ausschließlich aus den Eingangsdaten produziert und dabei die Eingangsdaten selbst nicht verändert.**

Diese Definition schließt die Rückwirkungsfreiheit oder auch *Seiteneffekt-Freiheit* ein: Es werden keinerlei Daten geändert. Es werden nur neue Daten als Funktionsergebnis produziert. Das Wort **ausschließlich** im obigen Text enthält und betont, dass die Funktion selbst keine eigenen Daten kennt, also *stateless* ist. Diese Definition ist umfassend, so dass es eigentlich keiner sekundären Definition bedarf (anders als im Abschnitt Objektorientierung).

Ob die Funktion dabei direkt aufgerufen wird, oder in einem Objekt gekapselt damit variabel aufgerufen werden kann, mag als sekundäre Definition angesehen werden und ist eine zweckdienliche Eigenschaft einer Funktionalen Programmiersprache. Diese ist allerdings auch schon mit der Objektorientierung gegeben, denn: Die Funktion kann die einzige überladene Operation eines Basisobjekt-Types sein, das dem Funktionstyp entspricht. Genauso funktioniert die Funktionale Programmierung in der OO-Sprache Java mit den sogenannten Lambda-Expressions.

3.1 Vorteile der Funktionalen Programmierung, Nachteile

Die Seiteneffekte, die in der Objektorientierung bei entsprechender möglicherweise nachlässiger Programmierung auftreten können und erheblich stören, werden durch den Funktionalen Ansatz vollständig ausgeschlossen.

Dies bring eine Sicherheit im Programmdesign. Man kann sich sicher sein, dass nach Aufruf einer Funktionalen Operation die Daten eben nicht geändert sind, sondern nur das produzierte Funktionsergebenis stellt neue Daten dar.

Der zweite Aspekt ist: Es werden immer die gleichen Daten bei gleichen Input-Daten erzeugt. Das kann Optimierungen ermöglichen. Wenn festgestellt werden kann, das unveränderte Input-Daten vorliegen, dann kann auf eine möglicherweise aufwändige neue Berechnung verzichtet werden. Dies kann ein sehr wesentlicher Aspekt sein.

Nachteile: Rein funktionale Programmiersprachen mögen etwas erschwerend bei Variablen sein, da diese defacto nicht änderbar sind. *(Dass sie dennoch häufig als Variable bezeichnet werden, ist irritierend)*.

Die Vorteile der Funktionalen Programmierung in bestimmten Bereichen sinnvoll kombiniert mit den anderen Programmierparadigmen (Objektorientiert, Event-gesteuert) erscheint zweckmäßig als allgemein anerkannte und verwendete Programmierherangehensweisen.

3.2 Zwischenwerte speichern in Funktionaler Programmierung

In der funktionalen Programmierung werden immer Werte neu erzeugt. Man kann die Funktion als Ganzes in der Programmierung direkt als Argument einer weiteren Funktion verwenden. Dabei kommt es oft und gewollt zu rekursiven Aufrufen. Man kann aber auch das Funktionsergebnis in einer Variablen speichern, die dann von einer folgenden Funktion verwendet wird. Das ist immer noch "*stateless*", wenn die Variable nicht in einer Schleife wiederverwendet wird sondern tatsächlich nur in der Vorwärtsberechnung als temporärer Zwischenspeicher dient.

Beispielsweise sollen aus 2 gegebenen Werten, die einen Punkt in der Fläche repräsentieren können, der normierte Vektor berechnet werden. Wenn man im weiteren Verlauf noch den Winkel benötigt, kann man dies über atan2(im, re) erledigen. In einer Aufrufzeile sieht das in Java wie folgt aus:

```
ComplexDouble(Math.cos(Math.atan2(b,a)), Math.sin(Math.atan2(b,a)));
```

Nun kann man vermuten oder hoffen, dass der Compiler die etwas aufwändigere Berechnung des atan2 selbständig optimiert, beide mit den selben Argument gerufen liefert das gleiche Ergebnis. Das lässt sich optimieren wenn die Funktionale Natur dieser Aufrufe bekannt ist. In Objektorientierten Funktionsaufrufen können diese Optimierungen vom Compiler nicht ausgeführt werden, denn es kann sein, dass beide Aufrufe nacheinander ausgeführt verschiedene Ergebnisse liefern, da Zwischenwerte gespeichert werden. Dies demonstriert eben gerade den Vorteil des Funktionalen Ansatzes.

Dennoch können die Zwischenwerte gespeichert werden, was genau das gleiche ist, aber besser lesbar und besser testbar:

```
final double angle = Math.atan2(b,a);
ComplexDouble(Math.cos(angle), Math.sin(angle));
```

In Java ist die Zwischenvariable als `final` gekennzeichnet, bedeutet dass sie nicht überschrieben werden kann und passt zum Paradigma der Funktionalen Programmierung: Keine Zustände speichern. Die Variable ist lediglich ein Zwischenwert.

Die hier gerufene Funktion `ComplexDouble(re, im)` ist eigentlich ein Constructor und erzeugt neue Daten, die Repräsentation eines komplexen Wertes (komplex im mathematischen Sinn mit real- und imaginär-Anteil). Man kann das Ergebnis dieser Funktion wieder direkt als Argument in einem Aufruf verwenden oder ebenfalls zwischenspeichern:

```
  final ComplexDouble result = new ComplexDouble(Math.cos(angle), Math.sin(angle));
```

Wen die Java-sprachtypische Konstruktion mit `new` stört: Man kann auch eine statische Routine aufrufen, die intern `new` aufruft.

```
  final ComplexDouble result = ComplexDouble.build(Math.cos(angle), Math.sin(angle));
```

Das sind letztendlich nur Ausführungsdetails.

Hier wird wieder das typische `final` verwendet. Im Zusammenhang mit der Constructor-Eigenschaft wird tatsächlich rein Funktional programmiert: Es entstehen mit dem Aufruf der Funktion neue Daten, die nur zwischengespeichert und nicht als State verwendet werden. Die Programiersprache Java ist hier hilfreich, da neue Daten recht schnell generiert werden können, dies auch üblich ist, und bei weiterer Nichtverwendung vom Garbagecollector für die Anwenderprogrammierung aufwandsfrei wieder entsorgt werden *("was in der Mülltonne ist, ist weg und interessiert nicht mehr")*.

Wie ist dies nun in C bei fast-Realtime-Anforderungen. Auch dort muss gegebenenfalls ein Winkelwert aus einem Vektor berechnet und nach Neuberechnung gespeichert werden (beispielsweise in *Feldorientierter Regelung*). Die Idee des Funktionalen Programmieransatzes sollte wegen der Vorteile auch hier verwendet werden.

Nun ist das Schreiben auf eine vorher angelegte Instanz immer noch stateless, wenn diese Instanzdaten stateless verwendet werden. Es ist nur nicht mehr formalistisch erkennbar:

```
  const double angle = atan2(b,a);
  ComplexDouble result = {0};
  set_ComplexDouble(&result, cos(angle), sin(angle));
```

Die `result`-Struktur kann auch innerhalb anderer Daten instanziiert sein und nicht wie hier dargestellt lokal (im Stack). Damit ist es möglich, dass das `result`-Ergebnis zur Berechnung selbst verwendet wird, das wäre also statefull und nicht Funktional. Wird es aber nicht verwendet, liegt immer noch Funktionale Programmierung vor. Dies muss per Doku (Comment) erklärt werden, über aufwändigere Codeanalyse wäre es auch im Quellcode erkennbar, wenn keine Tricks in C eingebaut sind. Das ist aber ein Problem der Programmiersprache bzw. des Programmierstils und nicht des Paradigmas.

3.3 Rekursive Algorithmen

Die Funktionale Programmierung wird meist eng mit der Anwendung von Rekursionen in Verbindung gebracht. Dazu ein Zitat aus der englischen Wikipedia https://en.wikipedia.org/wiki/Functional_programming#Recursion, abgerufen am 2019-07-31: "*Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, ...*"

Da in der Funktionalen Programmierung keine Zwischenwerte gespeichert werden, ist die rekursive Schreibweise tatsächlich notwendig. Die Frage ist aber, ob man immer damit glücklich wird. Folgend gibt es eine Funktion, die den maximalen Wert zweier Werte zurückgibt (Java):

```
  private static int retmax(int currmax, int value) {
    if(value > currmax) return value;
    else return currmax;
  }
```

Angewendet auf eine feststehende Inputmenge mit konsequentem rekursiven Aufruf sieht die Maximalbestimmung aus mehreren Werten wie folgt aus:

```
  public static int searchMax() {
```

```
int[] values = { 2,5,4,9,3,6 };
final int max1 = retmax(retmax(retmax(retmax(retmax(retmax(0, values[0])
, values[1]), values[2]), values[3]), values[4]), values[5]);
```

Das `retmax(...)` wird also rekursiv geschachtelt gerufen, jeder Aufruf bekommt einen Wert der Inputmenge übergeben, der andere Wert ist aus der aus dem rekursivem Aufruf vorher bestimmte größere Wert. Das Ergebnis ist korrekt, 9. Die Verfolgung im Debugger macht keinen Spaß.

Diese Funktionale Operation angewendet in einer imperativen for-each-Schleife debuggt sich viel einfacher:

```
int[] values = { 2,5,4,9,3,6 };
int max2 = 0;
for(int value: values) {
  max2 = retmax(max2, value);
}
```

Es gibt hier in der Aufrufumgebung eine state-Variable `max2`, die das bisher bestimmte Maximum enthält. Diese wird in der Schleife wiederverwendet. Die aufgerufene Funktionale Operation `retmax(...)` ist selbst stateless und seiteneffektfrei, was für das Verständnis des Programms gut ist - eine zielführende Kombination von Funktionaler und Imperativer Programmierung.

Rekursionen sind in der Programmierung teils das geeignete Mittel, aber nicht immer. Es gilt jedenfalls die Regel: "*Rekursion sollten dann in der Programmierung angewendet werden, wenn das Problem selbst rekursiv ist.*", Rekursionen sind zumeist gegen etwas aufwändiger zu formulierende Schleifen austauschbar. Dazu müssen Daten gespeichert werden, die bei Rekursionen ohne Zusatzaufwand in lokalen (Stack-) Variablen stehen. Rekursionen sind im Test aufwändiger, weil man die Daten, die zum Ergebnis führen, im Stack suchen muss, anstelle einfacher in den aktuell vorhandenen Daten. Dafür ist der Aufwand, die Daten für die Schleifen für nicht rekursive Ausführung zu bilden, höher. Das Thema Rekursion ist weiter gefächert auch außerhalb des Themas *Funktionale Programmierung*. In der Kombination von Funktionaler mit traditioneller (imperativer) Programmierung reduziert sich das Thema Rekursion auf das eigentliche. Siehe auch Vorabschnitt 3.2 Zwischenwerte speichern in Funktionaler Programmierung.

Ziel dieses Artikels ist es, auf die Verwendung und Verwendbarkeit der Funktionalen Programmierung in allgemeinen programmtechnischen Anwendungen die auch imperativ und oft objektorientiert sind, hinzuweisen. Im Kontext der Betrachtung rein Funktionaler Programmierung sehen diese Themen anders aus. Das wird hier nicht behandelt, dies nur als deutlicher Hinweis der Abgrenzung.

3.4 Einfache Beispiele für Funktionale Programmierung in bekannten Programmiersprachen

Die mathematischen Funktionen sin, cos, abs, sqrt und wie sie alle heißen sind funktionale Programmierung, in allen Programmiersprachen.

Ausnahme: Wenn etwa in speziellen embedded Implementierungen ein sin-Ergebnis vom gepspeicherten Wert eines vorigen Aufrufes abhängt, dann ist diese sin-Berechnung nicht funktional. Man kann mit einer solchen Lösung Rechenzeit sparen und die Genauigkeit erhöhen, weil eine Berechnung vom letzten Wert ausgehend Iterationsschritte einsparen kann. Das kann sehr praktisch sein, da eine trigonometrische Berechnung komplex ist und für eine sehr schnelle Abtastzeit sich dieser Tricks bedienen kann. Das trifft aber nicht für die Standard-mathematischen Funktionen zu.

Wenn in C oder Java oder usw. formuliert wird:

```
int calcXYZ(float x, float y, int z){
  return x*y*table[z];
}
```

dann ist dies funktionale Programmierung, wenn die `table const` ist.

Die funktionale Programmierung begegnet uns also öfter als geläufig.

3.5 Funktionale Programmierung mit nur einem Konstruktor einer objektorientierten Klasse

Die Forderung der Funktionalen Programmierung ist, dass die Ergebnisdaten erzeugt werden sollen, nur aus den Eingangsdaten. Wenn insgesamt funktionale Programmierung verwendet wird, oder wenn diese Ergebnisdaten als "von funktionaler Programmierung als unverändert erzeugt" weiterverarbeitet werden sollen, egal ob im funktionalen Kontext oder nicht, dann dürfen die Daten selbst danach nicht mehr geändert werden. Für den einfachen Fall der numerischen Funktionen kann das so interpretiert werden, dass eine Neubelegung der Variablen, der das Funktionsergebnis zugewiesen wurde, außerhalb der Funktionalen Programmierung steht:

```
float y = sin(x);
```

Die Belegung von y ist funktional, nicht aber y selbst.

```
y = 234.5;
```

steht also außerhalb der Eigenschaft der sin-Funktion, Funktional zu sein.

Wenn nicht einfache numerische Werte Argumente und Funktionswerte sind, sondern komplexe Datenstrukturen, dann leistet in Java ein Konstruktor genau dies, wenn alle Elemente der Klasse final sind:

```
final class FunctionExample {
  public final double m, rad;
  FunctionalExample(double re, double im){
    m = Math.sqrt(re*re + im*im);
    rad = Math.atan2(im, re);
  }
}
```

Diese class enthält nur den Konstruktor, keine wetiteren Operationen und keine mögliche Ableitung. Mit dem Aufruf des Konstruktors

```
FunctionExample result = new FunctionExample(a,b);
```

wird der Ergebnisreferenz ein neu erzeugtes Datenobjekt zugewiesen, dessen Werte unveränderlich (weil final) sind. Die Inputargumente (es wären hier ebenfalls Referenzen möglich) werden auch nicht geändert. Folglich ist dies rein Funktional. Wenn die Referenz, die das Ergebnis referenziert, nachfolgend geändert wird, dann ist dies außerhalb dieses funktionalen Kontexts. Man kann die Ergebnisreferenz ebenfalls `final` definieren, um dem vorzubeugen.

3.6 Funktionale Programmierung mit Lambdas ab Java 8

Die Grundlage ist die Möglichkeit, eine Funktion syntaktisch einfach zu bestimmen. Den Rahmen dafür bildet das

```
/**A functional interface is necessary for a lambda expression.
 * It represents the type of expression.
 * It does not define how to do, only the basic requirements. */
@FunctionalInterface interface Dosomething {
  MyData doit(int val, MyData arg);
}
```

Die Annotation `@FunctionalInterface` hat Kommentarwirkung und regelt syntaktisch, dass im Interface nur eine Operation definiert werden kann. Man kann die Annotation grundsätzlich auch weglassen.

Hier wird eine Funktionsignatur definiert, die eine int-Wert und ein Datenobjekt entgegennimmt und eine Instanz zufällig vom gleichen Typ (kann auch anders sein) zurückliefert. Verglichen mit C ist dies vergleichbar mit der Typdeklaration einer Funktion für einen Funktionszeiger:

```
typedef MyData* DoSomething(int val, MyData* arg);
```

Der Funktionszeiger selbst wird dann in C wie folgt definiert:

```
DoSomething* aFunctionPointer;
```

Hinweis: In C oft gebräuchlich ist die Schreibweise

```
typedef MyData* (DoSomething*)(int val, MyData* arg);
```

direkt für den Funktionszeiger, damit man bei dessen Anwendung nur schreiben braucht:

```
DoSomething aFunctionPointer;
```

So steht es meist in den Lehrbüchern. Bei dieser Scheibvariante sieht man aber der Zeigerdefinition für den Funktionszeiger genau dies nicht an. Man weiß nicht, dass dort ein Zeiger (auf irgend etwas, siehe typedef) verwendet wird. Daher ist die obige Variante in C die gegebenenfalls eigentlich bessere, aber weniger bekannt.

Die Anwendung des obigen Interfaces sieht dann klassich, ohne Nutzung der Lambda-Möglichkeiten, in Java wie folgt aus:

```
 /**Old style definition of a functionality with an implicit implementation
  * of an interface operation: */
  Dosomething variant1 = new Dosomething() {
    @Override public MyData doit(int val, MyData arg) {
      MyData ret = new MyData(arg.x1 * val, arg.x2); return ret;     }};
```

Verglichen mit C wäre das etwa:

```
MyData* doSomethingVariant1(int val, MyData* arg) {
  MyData* ret = (MyData*)malloc(sizeof(MyData));
  ret->x1 = arg->x1 * val; ret->x1 = arg->x2;
  return ret;
}
.....
DoSomething variant1 = doSomethingVariant1; //set the function pointer.
```

Insoweit ist in C schon immer realisierbar, was in Java auch schon immer möglich ist. Nun aber zu den neuen Lambda-Möglichkeiten. Dies ist eine Frage der Syntax:

```
   /**It is the same, but shorter written. That is a lambda expression. */
   Dosomething variant2 = (val,arg) ->
               { MyData ret = new MyData(arg.x1 * val, arg.x2); arg.x2 = 0; return ret; };
```

Die erste noch ausführliche Schreibweise definiert die Implementierung des `@FunctionalInterface Dosomething` noch als extra Anweisung, aber einfacher geschrieben. Die Argumente der Operation (Function) wird vor dem `->` angegeben, bei einem Argument als typischen Fall kann die Klammerung weggelassen werden. Der Vorteil, es braucht nicht eine extra anonyme Definition der Interface-Implementierung. Ein Unterschied: Die anonyme Interface-Implementierung ist eine innere Klasse mit `this$0`-Pointer auf die umgebende Instanz. Die neue Lambda-Schreibweise hat diesen Zeiger nicht (das ist ein Detail).

Es gibt eine verkürzte Schreibweise:

```
  /**For only one statement the { return } can be omitted */
   Dosomething variant3 = (val,arg) -> new MyData(arg.x1 * val, arg.x2 = 0);
```

Diese verkürzte Schreibweise geht also für die typischen Anwendungsfälle und ist dann auch elegant in einem Aufruf verpackbar:

```
  /**A lambda expression can be immediately written as argument.
   * The argument contains 'how to do'. */
   d4 = routine((val,arg) -> new MyData(arg.x1 * val, arg.x2), 23, userArg);
```

Wichtig für das Funktionieren dieser Schreibweise ist es, dass an der Verwendungsstelle der Typ, hier `Dosomething`, bekannt ist. Das ist aber immer der Fall in den typischen Anwendungsfällen. Die `routine` ist wie folgt definiert:

```
 MyData routine(Dosomething dosomething, int val, MyData arg) {
   return dosomething.doit(val,arg);
 }
```

Der Argumenttyp besorgt die syntaktische Ordnung.

Die Anwendung dieser Schreibweise ist dann häufig mit den Container-Streams verbunden. Das ist ebenfalls eine neue Möglichkeit ab Java-8 und darf nicht mit den IO-Streams verwechselt werden:

```
  List<MyData> list = new LinkedList<MyData>();
   for(int x = 2; x < 6; ++x) {             //build a container as example
     list.add(new MyData(x, 2*x+3));
   }

   list.stream().forEach(d -> d.x2 = 333);   //work with container data
```

In diesem Beispiel ist aber eine grobe Verletzung des Funtionalen Paradigma eingebaut, ein Seiteneffekt. Es wird als Beispiel die Variable `x2` aller Elemente im Container auf `333` gesetzt. Das ist ein Seiteneffekt, der mitnichten dem Paradigma der Funktionalen Programmierung entspricht.

Es ist nun die Frage zu stellen: Ist dies Funktionale Programmierung? Klare Antwort: Nein, auch wenn alle Literaturstellen dies als Ansatz der Funktionalen Programmierung in Java darstellen.

Es wurde lediglich eine Schreibweise gefunden, wie sehr einfach ein Code-Snippet benannt werden kann. Die Anwendung auf die neuen `stream()` ist interessant. Interessant ist auch der Aspekt der Parallelisierbarkeit der Verarbeitung von Containerinhalten. Das hat aber mit Funktionaler Programmierung nicht so sehr viel zu tun. Wie in der C-Syntax schon gezeigt, es handelt sich lediglich um eine einfache Möglichkeit (einfacher als in C), eine Funktion als Code-Snippet zu

schreiben, oft elegant, kurz und durchaus auch verständlich. Ein Lob an Java, aber es ist keine Funktionale Programmierung. Denn:

* Insgesamt getestet an Java Version 8.211 von Oracle, Man kann in die Funktionen, die mit `@FunctionalInterface` gekennzeichnet sind, beliebig Seiteneffekte hineinbauen. Es wird syntaktisch nicht verhindert und nicht geprüft.

* Damit ist dann auch das geforderte stateless umgehbar: Indem in den Argumente, die auch wieder rückgegeben werden können, Variablen überschrieben werden können, kann man sich Zustände merken.

Damit sind die wichtigsten zwei Merkmale der Funktionalen Programmierung nicht unterstützt.

Die wesentlichste Syntaxmöglichkeit der Verhinderung von Seiteneffekten fehlt in Java von Anfang an, aber in C/++ ist sie möglich: `const*`-Kennzeichnung von Referenzen. Zusammen mit der Java-Schreibweise der sogenannten Lambda-Ausdrücke wäre dann eine wirkliche Funktionale Programmierung möglich. Siehe Kapitel Error: Reference source not found Error: Reference source not found, Seite Error: Reference source not found

3.7 Funktionale Programmierung in der Automatisierungstechnik

In der Automatisierungstechnik ist nach der Norm IEC 61131-3 eine Funktion (`FUNCTION`) vollständig funktional, wenn die `FUNCTION` keine `VAR_IN_OUT` benutzt. Letztere sind von außen zugewiesene Zustandsvariable.

* Auf Variable mit `VAR_IN` definiert wird nur lesend zugegriffen. Das betrifft auch zusammengesetzte Variable (mit `TYPE` definiert)

* Die Variable mit `VAR_OUTPUT` definiert sind die Variable für das Funktionsergebnis. Aus Sicht der `FUNCTION` ist es lediglich die Ergebnisablage. Es kann sich im Aufrufkontext um `VAR_TEMP` handeln, lediglich Zwischenspeicher für die Verwendung in den folgenden Aufrufe. Dann ist auch die Nutzung einer `FUNCTION` Funktional. Nur wenn die Variable beim folgenden Aufruf wieder als Input auftreten, handelt es sich um Zustandsvariable.

* In der Kombination von `FUNCTION` mit `FUNCTION_BLOCK` ist die einzelne `FUNCTION` ohne `VAR_IN_OUT` dennoch als Funktionale Programmierung betrachtbar, nicht jedoch das gesamte Programm.

Ein `FUNCTION_BLOCK` hat dagegen eigene Zustandsvariable. Es wird damit das stateless-Kriterium jedenfalls nicht erfüllt. Es handelt sich in diesem Fall um eine Datenflussorientierte Programmierung.

3.8 Notwendige statefull Programmierung

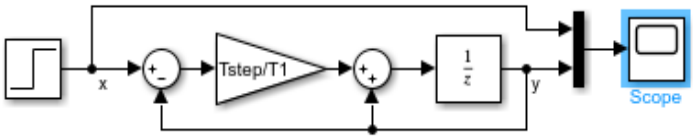
Man kann im Kontext der Beschäftigung mit Funktionalen Programmierung schnell vergessen, dass zustandsbehaftete (statefull) Dinge teils immanent sind und damit programmtechnisch geboten. Als Beispiel soll das einfache T1-Glied in der Regelungstechnik dienen (*Trägheitsglied erster Ordnung*). Es beschreibt ein Ausgangssignal, das dem Eingangssignal etwas verzögert mit teilweiser Bewahrung des Altwertes folgt. Jede Änderung, beispielsweise einer Temperatur im Wasserkocher, folgt in der einfachsten Form diesem Verhalten. Es ist eine Differenzialgleichung, in der der Altwert selbstverständlich eine tragende Rolle spielt. Numerisch formuliert kann das T1-Glied wie folgt werden:

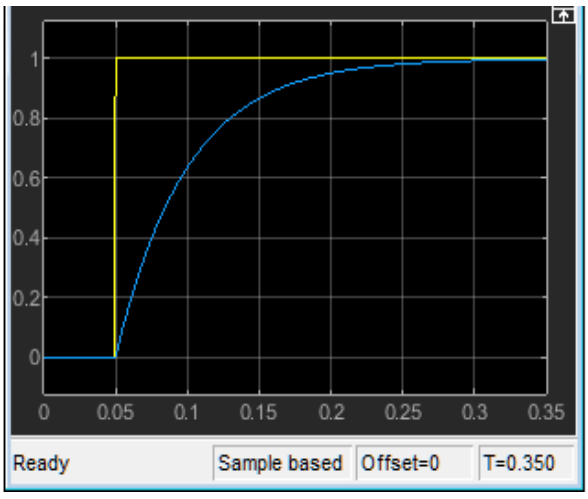
```
  class T1_Ctrl {
```

```
  double y;
  double kT1;

  T1_Ctrl(double T1, double Tstep){
    kT1 = T1 < Tstep ? 1.0 : Tstep/T1;

  double calc(double x) {
    y += kT1 * (x - y);
    return y;
  }
}
```

Im Konstruktor ist `Tstep` die Abtastzeit, die Werte `x` müssen aus dem realen Prozess äquidistant dieser Abtastzeit folgen. Wird der Algorithmus also in Realtime angewendet, dann muss die Reaktionszeit auf Messwerte eingehalten werden. Man kann aber insbesondere für Simulationen diesen Algorithmus auch ohne Realtime-Anforderungen nutzen. T1 als Konstruktor-Argument ist die gewüschte Trägheit, die Zeit, in der eine sprunghafte Änderung zu etwa 63% im Ausgangswert wiedergefunden wird. Wenn beispielsweise bei einer Änderung der Kohlendioxidbesteuerung nach 10 Jahren im Schnitt 63% der Einfamilienhäuser ihre Feuerung auf eine modernere Form als Ölheizung umgestellt haben, dann ist dies die Zeitkonstante. Die Berechnung des kT1-Faktors ist vereinfacht und gilt in Näherung nur für große Werte der Zeitkonstante gegenüber der Abtastzeit. Eine genauere Berechnung die auch bei kleinen Abtastzeiten gilt, braucht eine e-Funktion, hier nicht dargestellt.

In der Operation `class_T1_Ctrl#calc(...)` wird nun zweimal auf die Zusatndsvariable `y` zugegriffen, der zweite nicht direkt sichtbare Zugriff erfolgt über den `+=`-Operator. In einem zugehörigen Strukturbild aus Simulink sind ebenfalls zwei Rückführungen zu erkennen:





Für den Ansatz der funktionalen Programmierung muss das Programm etwas umgeschrieben werden. Es ist eine Interpretationsfrage, ob der Altzustand y als Zustand aufgefasst wird, oder als Berechnung eines neuen Wertes unter Berücksichtigung eines gegebenen (alten) Wertes. Der letzte Ansatz ist funktional und sieht wie folgt aus:

```
class T1_Ctrl_Functional {
  final double kT1;

  T1_Ctrl_Functional(double T1, double Tstep){
    kT1 = T1 < Tstep ? 1.0 : Tstep/T1;
```

```
    }

    double calc(double x, double y_last) {
      return ylast +  kT1 * (x - ylast);
    }
  }
```

Beim Aufruf kann dann beispielsweise ein Array mit den jeweiligen Werten gefüllt werden, es wird also wie Funktional gefordert ein Wert niemals überschrieben.

```
T1_Ctrl_Functional t1 = new  T1_Ctrl_Functional(500, 1);
double[] y_array = new double[10000]; //considered time
y_array[0] = 0.0; //start value
for(int ix = 1; ix < y_array.length; ++ix) {
  y_array[ix+1] = t1.calc( x, y_array[ix]);
}
```

Dies kann auch rekursiv formuliert werden:

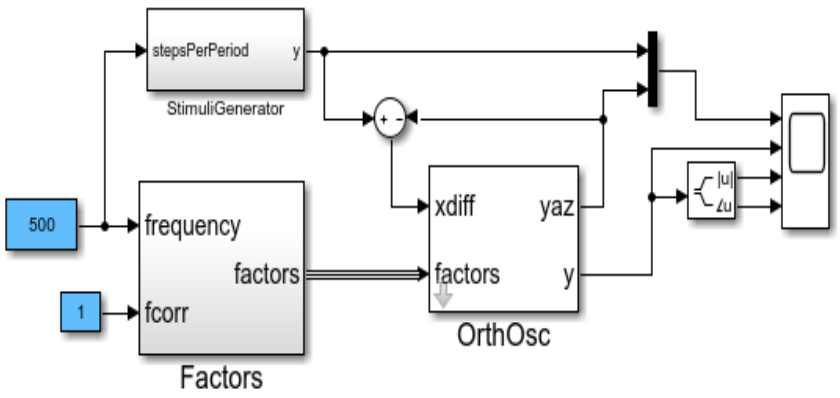
```
double calcRecursive(double x, double y_last) {
  return Math.abs(x-y_last) < 0.00001
       ? y_last
       : calcRecursive(x, calc(x, y_last));
}
```

Es muss hier ein Abbruchkriterium geben, in diesem Fall ist es die genügend genaue Erreichung des Endwertes. Der exakte Endwert wird mathematisch nie erreicht (asymptotisch). Die Zwischenwerte werden jeweils im Stack aufgehoben, statt im Array wie im Schleifen-Beispiel. Man kann beispielsweise die Zwischenwerte ausgeben. Doch dazu braucht man wieder ein Statement, das sehr imperativ aussieht. Ansonsten hat das Beispiel aber wenig praktischen Nutzen.

Es stellt sich daher die Frage nach den Grenzen der Funktionalen Programmierung. Es wäre sehr vereinfacht, anbetrachts der Tatsache das Zustandsvariable teilweise notwendig sind, auf die Funktionale Programmierung ganz zu verzichten. Richtig ist also die gute Mischung.

4. Datenflussorientierte Programmierung

Es gibt ein weiteres Programmierparadigma, dass der Funktionalen Programmierung etwas ähnelt, aber in Informatikerkreisen wenig diskutiert wird. Die Datenflussorientierte Programmierung ist aber allbekannt und verbreitet in der grafischen Programmierung mit Funktionsblöcken:



Das rechtsstehende Bild zeigt ein Simulink-Modell, in dem Funktionsblöcke verschaltet sind. Die Pfeilrichtungen an den Verbindungen geben den Datenfluss an. Dass Modul *OrthOsc* bekommt also Daten vom Modul *Factors* geliefert und liefert seinerseits Daten in diesem Beispiel letzlich nur an den Scope, zur Ergebnisbetrachtung, da dieses eine Testschaltung ist. Im Vorkapitel ist ebenfalls ein Simulink-Modell für das T1-Glied dargestellt, dass den Datenfluss nur für Grundelemente und numerische Grunddatentypen darstellt. In diesem Fall ist die *factors*-Verbindung ein zusammengesetzter Datentyp, in Simulink als Bus dargestellt.

Charakteristisch für diese datenflussorientierte Verbindung der Module ist, dass ein Rückschreiben gegen die Pfeilrichtung nicht vorgesehen ist. Das ist nicht unbedingt so, denn mit dem Datenfluss lassen sich auch Handle vorwärts reichen, die ein Rückschreiben in die damit verbundene Daten (über die Handle-Verwaltung) gestattet. Ein solches System mit Simulink ist in ESE2017: Grafische Objektorientierte C-Programmierung mit Simulink beschrieben, von mir auf dem ESE-Kongress 2017 vorgestellt. Dieses dort vorgestellte Prinzip nutzt zwar die Darstellungsmechanismen des Datenflusses, entspricht aber nicht mehr dem Paradigma. Es muss daher sehr genau unterschieden werden, ob über einen vorwärts propagierten handle zurückgeschrieben werden kann und muss, oder ob der Handle lediglich den readonly-Zugang zu den Daten gestattet. Im letzterem Fall ist es nur eine Frage der Implementierung, ob das grafische gezeigte handle nicht doch eine Eventkommunikation zum weiterreichen der Daten ist, ähnlich wie bei der Funktionsblock-Kopplung in der IEC 61499.

Die Funktionsblock-Programmierung lässt sich natürlich auch in Zeilenprogrammierung darstellen, hier in Java auszugsweise:

```
float frequency = 500;
float fcorr = 1.0;
Factors factor_FBlock;
OrthOsc orthOsc_FBlock;
factors_FBlock.step(factors_FBlock , frequency, fcorr);
float xdiff = stimuliGenerator_FBlock.y - orthOsc_FBlock.yaz;
orthOsc_FBlock.step(orthOsc_FBlock, factors_FBlock.factors, xdiff);
```

4.1 Unterschiede zwischen Funktionaler Programmierung und Datenfluss

Wird der Datenfluss als nicht-rückschreibefähig verstanden, dann ist der einzige Unterschied zwischen Datenfluss und Funktionaler Programmierung derjenige, dass die Datenfluss-Programmierung in den FBlocks nicht stateless ist.

Da das stateless aber für einige Anwendungen eine zu starke Forderung ist, scheint es zweckmäßig, für ein gutes Design auf das Funktionale Prinzip Verhinderung von Seiteneffekten zu setzen, und ansonsten eigentlich Objektorientiert zu arbeiten.

Damit gewinnt aber die compilerformale Verhinderung des Rückschreibens, wie sie in C/++ mit `const*` und `const&` gegeben ist, bei Anwendung des Prinzips in Java an grundlegender Bedeutung. Damit ist eine Forderung der Unterstützung von @ReadOnly im Java-Sprachkontext verbunden. Der Compiler muss dies prüfen.

4.2 Datenflussorientierte Programmierung in der Automatisierungstechnik

Wie schon im Kapitel 3.7 Funktionale Programmierung in der Automatisierungstechnik Seite 22 erwähnt stellen `FUNCTION_BLOCK`-Definitionen zwar keine Funktionale, wohl aber eine Datenflussorientierte Programmierung dar, genauso wie die FBlock-Verknüpfungen in den Function Block Diagrammen und auch in CFC-Grafiken (Continous Function Chart - Wikipedia. Die FunctionBlock-Grafiken aus IEC 61131-3 und ihre CFC-Erweiterung reihen sich in die FBlock-Grafiken in anderen Darstellungen wie Simulink ein. Die Unterschiede sind nicht prinzipieller Natur.

4.3 Datenflussorientierung und Objektorientierung

Man kann als Datenfluss eine Referenz aus einem FBlock einem anderen FBlock übergeben. Über die Referenz kann dann der empfangene FBlock auf interne Daten des anderen FBlock zugreifen. Das entspricht der Verbindung von Objekten mit einer Assoziation oder Aggregation.

Diese Herangehensweise ist insbesondere möglich, wenn die FBlocks intern direkt (in C) programmiert werden. Die Umsetzung der FBlock-Verbindung als 'Referenz' auf die tatsächliche Speicheradresse kann auf diesem Weg immer realisiert werden.

Die andere Möglichkeit ist die Eventkommunikation als Datenfluss.

5. const*- oder @ReadOnly-Referenzen als Sicherung gegen Rückschreiben / Vermeidung von Seiteneffekten bzw. gegen den Datenfluss

Im Kapitel 3.6 Funktionale Programmierung mit Lambdas ab Java 8 Seite 20 wurde dargestellt, dass die Programmierung in Java syntaktisch Seiteneffekte nicht verhindert. Was fehlt, ist eine Kennzeichnung der Referenzen mit so etwas wie `@ReadOnly`, semantisch eindeutig.

Seiteneffekte können zwar erklärtermaßen nicht programmiert werden. Besser ist es aber, wenn durch entsprechende Zugriffsmechanismen wie private-Kapselung der Compiler überprüft, ob dass es keine Seiteneffekte, sprich Rückschreiben über Referenzen gibt. Aus einem begrenztem Kontext heraus, dem der Klasse oder der einzelnen Operation, muss erkannt werden können, dass es programmtechnisch nicht möglich ist, über referenzierte Daten zu verändern. Die Erklärung, dass würde nicht geschehen, reicht nicht aus. Denn: Programme können fehlerhaft sein. Wenn compilertechnisch klar ist, dass es kein Rückschreiben geben kann, dann ist eine klare Aussage oder im Negativfall eine Fehlersuche sehr viel einfacher.

C und C++ ermöglicht eine solche Aussage mit der const*-Kennzeichnung von Referenzen. Allerdings gibt es dort die Hintertür des cast von Referenzen. In Java ist diese Aussage auf Referenzen bezogen derzeit leider nicht möglich, was als wesentlicher Nachteil zu empfinden wäre.

5.1 const in C und C++ als Rückschreibeverhinderung über Referenzen*

In C, auch in C++ gibt es das. Man kann eine Zeigerdefinition mit `const*` bezeichnen. Das const bezieht sich hier nicht auf die Variable selbst (was in Java etwa `final` wäre) sondern auf die Rückschreibbarkeit auf die referenzierten Daten. Dazu aber etwas ausführlicher, da sonst möglicherweise falsch verstanden:

Folgend wird ein const Array definiert. Das const-Array kann sich in einem readonly-Speicherbereich befinden.

```
const int array[] = { 1,2,3,4,5};
```

Wenn man nun einen Zeiger auf das Array will, dann muss verhindert werden, dass der Compiler Schreibbefehle aktzeptiert. Die seit K&R-C seit 1970 eingebürgerte Schreibweise wäre:

```
const int* ptr_array = &array;
```

Klar, `const` steht beidemal ganz links. Aber nicht die Variable `ptr_array` ist const, sondern der referenzierte Inhalt. Man kann also ptr_array auch erneut belegen. Das war in der Überlegung um 1970 aber nicht wichtig zu kennzeichnen. Es geht aber:

```
const int* const ptr_array = &array;
```

Das zweite `const` bezieht sich nun auf die Pointervariable selbst. Semantisch einfacher, syntaktisch immer möglich ist es zu schreiben:

```
int const* ptr_array = &array;
```

In diesem Fall wieder mit änderbarer Variable `ptr_array`. Besser erkennbar ist, dass das `const` eine Eigenschaft der Referenzierung ist, die mit dem `*` ausgedrückt wird. Beides steht zusammen.

Aus Sicht von 2019 ist nun das mögliche Rückschreibeverbot über eine Referenz auf sonst beschreibbare Daten wichtiger. Man kann selbstverständlich schreiben:

```
const int array[] = { 1,2,3,4,5};
int* ptrWrArray = &array;
int const* ptrRdArray = &array;
```

Man hat hier einen Schreibzeiger und einen Nur-Lese-Zeiger auf die selben Daten. Diese Zeigerdefinitionen können genauso in den Argumenten von Funktionen verwendet werden:

```
int myOperation(int const* ptrData, int ix) {
  return ptrData[ix];
}
```

Wenn man einen Seiteneffekt unerwünscht einbaut, dann gibt es einen Compilerfehler:

```
int myOperation(int const* ptrData, int ix) {
  ptrData[0] += 1; //should count accesses, but compiler error!
  return ptrData[ix];
}
```

Die `const*`-Schreibweise in C und auch C++ verhindert also Seiteneffekte per Compilerfehlermeldung. Wie bekannt kann man in C zwar tricksen:

```
int myOperation(int const* ptrData, int ix) {
  ((int*)ptrData)[0] += 1; //should count accesses, no compiler error!
  return ptrData[ix];
}
```

Aber undokumentierte casting sollten beim Review auffallen und sind bei einer entsprechenden Sicherheitslevel-Programmierung verboten.

Wichtig ist, die `const*`-Kennzeichnung bezieht sich auf die einzelne Referenz, nicht auf die Daten als solche. In einem Kontext kann man so Daten modifizieren (ohne const, dort wo es notwendig ist), in einem anderen Kontext ist die Modifikation per Funktionsheader ausgeschlossen. Das wäre dann bei einer Funktionalen oder Datenfluss-Funktion.

Wenn eine Funktion ausschließlich mit `const*`-Referenzen arbeitet, dann ist sie für sich betrachtet stateless, da sie nirgends zurückschreiben kann. Kurzer Hinweis: Skalare Argumente brauchen diese Kennzeichnung nicht, da ihre Änderung in der Funktion sich nicht auf die Originale bezieht (*call per value*).

Wichtig weiterhin: Einmal `const*`, immer `const*`. Man kommt in C/++ nicht von der `const*`-Definition einer Referenz weg, außer dem direkten casting:

```
const int constArray[] = { 1,2,3,4,5};
const int wrArray[5] = { 0 };
int* ptrWrConstArray = &constArray;    //compiler error, wr to const data
int const* ptrRdArray = &constArray;  //ok
int* ptrWrArray = & wrArray;          //ok
int const* ptrRdonWrArray = & wrArray;  //ok, cannot wr via this reference
int const* ptrRd2WrArray = ptrWrArray;  //ok, derived from non const, but readonly here.
int* ptr2RdArray = ptrRdArry;           //compilererror cannot get non const from const
```

Dieses System ist wasserdicht.

Selbst der this-Zeiger in C++ ist mit `const*` definierbar, man muss das const dann hinter die schließende Klammer schreiben, da this implizit ist:

```
class MyClass {
  void myWrFunction(int args);
  void myReadOnlyFunction(int args) const;
  .....
```

Die C++-Funktion `myReadOnlyFunction(...)` entspricht damit der Funktionalen Programmierung. Sie kann auch in den eigenen Instanzvariablen keine states speichern.

Wenn die Referenz auf die Instanz vom class-Typ `const*` ist, dann können damit nur const-Methoden gerufen werden:

```
void myFunctional(MyClass const* refc) {
  refc->myReadOnlyFunction(234); //ok
  refc->myWrFunction(999);       //compiler error
```

Also hat C/++ die notwendigen syntaktischen Möglichkeiten, sowohl Funktional als auch Datenflussorientiert zu programmieren. Java aber nicht.

5.2 @ReadOnly Annotation als Rückschreibeverhinderung in Java ?

In Java könnte man selbiges erreichen mit einer `@ReadOnly` Annotation mit selbigen Eigenschaften wie `const*` in C. Das `@ReadOnly` muss vom Java-Compiler überprüft werden:

* Annotation von `@ReadOnly` an Referenzen
* Fehler bei Zuweisung einer `@ReadOnly`-Referenz auf passend typisierte, aber nicht @ReadOnly.
* ok bei Zuweisung einer nicht-`@ReadOnly`-Referenz auf eine `@ReadOnly`-Referenz.
* Fehler bei Schreiben auf Daten über eine `@ReadOnly`-Referenz
* Annotation von `@ReadOnly` an Operationen einer Klasse.
* Fehler bei Aufruf einer Nicht-`@ReadOnly`-gekennzeichneten Operation über eine `@ReadOnly`-Referenz
* OK bei Aufruf einer `@ReadOnly`-Operation über eine nicht-`@ReadOnly`-Referenz.

Das const*-Thema hat man offensichtlich in Java vergessen, missachtet oder man war damals der Meinung, dass mit dem Interface-Konzept viel besser readonly-Zugriffe designbar sind. Das stimmt zwar, aber auch bei einem Interfacezugriff sieht man der Zugriffsroutine nicht an, dass sie keine Seiteneffekte hat. In Java fehlt das wichtige Pendant zum `const*`, das zweckdienlich mit einer `@ReadOnly`-Annotation geschrieben werden sollte. Man kann eine solche Annotation in Java selbst definieren, aber sie wird nicht vom Compiler überprüft, ist also (tote) Dokumentation. Möglich ist eine spezielle statische Codeanalyse für `@ReadOnly` zusätzlich und außerhalb der offiziellen Java-Tools.

5.3 Rückschreibeverhinderung in Java mit Get-Operationen ?

Grundsätzlich kann man mit passenden Get-Operationen in Java, genauso wie in C und C++ das Rückschreiben compilerüberprüft verhindern. Die betreffenden Referenzen dürfen dann von der Anwenderprogrammierung nicht erreichbar sein. Das ist in Java und C++ durch eine entsprechende `private`- oder `protected`-Kennzeichnung möglich. In C kann man solche Referenzen beispielsweise mit einem `_priv` als Suffix kennzeichnen oder auch nur mit einem Unterstrich als Suffix, und in allen Quellen ggf. per automatischer Codeanalyse kontrollieren, dass nur compilermoduleigene Variable mit _ oder `_priv` benutzt werden.

Es ist die Frage zu stellen, ist damit das Problem der compilergesicherten Programmierung gegen Rückschreiben gelöst. Antwort: "*Theoretisch Ja*".

Warum nur "*theoretisch*"? Weil der Aufwand, entsprechende Getter zu schreiben, etwas höher ist. Der einfache Zugriff auf die Referenz geht viel schneller. Folglich wird man den Aufwand für die Getter meiden, und diese theoretisch richtige Möglichkeit in vielen Fällen nicht nutzen. Damit ist die Sicherheit weg.

Beispiel:

```
int[] array = new int[12];
```

Wenn man meint, mit einem Getter

```
int[] getArray() { return array; }
```

die Sache zu erledigen, dann irrt man! Selbstverständlich kann man das Array ändern, als ungewollter, programmfehlerhafter Seiteneffekt:

```
int[] ref = obj.getArray();
int value = ref[2];  //access, ok
ref[2] = 125;        //change not ok!
```

Das Getter auf die Referenz bringt also nichts. Es muss folgende zwei Getter geben:

```
int getArrayLength() { return array.length; }
int getArrayValue(int ix) { return array[ix]; }
```

Mit diesen beiden Getter ist der Schreibzugriff verhindert. Man könnte nunmehr argumentieren, das zweite Getter könne nochmal den Index überprüfen und eine Index-Exception verhindern oder dergleichen. Das wäre dann schon ein Argument für diesen Getter. Aber nicht für den einfachen Zugriff mit regulärem Index.

Für eine class mit Sub-Referenzen darf nicht die Sub-Referenz per Getter zurückgegeben werden, sondern nach der gleichen Argumentationskette muss es entweder für alle mittelbar referenzierten Daten entsprechende Getter geben, die nur die Werte selbst zurückgeben. Das wäre sauber und konsequent, und schließt eine programmfehlerhafte Änderung von Daten aus. Es ist aber aufwändig. Die andere Möglichkeit ist, die referenzierte class mit einem Interface zu versehen, das nur lesende Operationen, also Getter, enthält. Über ein anderes Interface könnte dann geschrieben werden, wenn das aus einem anderen Kontext notwendig ist. Diese konsequente Programmierung ist in Java möglich, damit ist Java sicher gestaltbar gegen programmfehlerhaftes Rückschreiben. Aber der Aufwand ist hoch und wird oft nicht getrieben. Die Folge sind möglicherweise unsichere Programme.

Um den Aufwand für Getter niedrig zu halten, wird oft eine andere Möglichkeit angeboten: Automatische Erzeugung von Getter-Routinen, dann auch automatische Erzeugung von Interfaces nur mit Getter-Routinen. Diese automatisch erzeugten Programmteile haben aber einen anderen wesentlichen Nachteil: Das Argument für eine private-Kapselung ist auch, dass diese Programmteile innerhalb der class geändert werden dürfen, ohne dass dies nach außen Auswirkungen hat. Genau dieses wichtige Ansinnen wird aber von den automatisch erzeugten Getter-Routinen auf alle `private`-Daten vollständig unterwandert.

Es ist sehr viel einfacher, bestimmte Referenzen als `public` zu deklarieren, weil sie `public` sein sollen, und nur einen lesenden Zugriff zuzulassen, wie `const*` in C/++. Dieser Programmierstil wird viel eher angenommen, da es einfacher ist und der Compiler bei einem Fehlzugriff oder einer Mischung von `const*` und nicht-const `MyType*` sofort diese Stellen kennzeichnet. Ein Refactoring für nachfolgend eingeführtes `const*` ist nicht zu aufwändig. Dies spricht für die adäquate `@ReadOnly`-Kennzeichung in Java.

6. Eventgetriebene Programmierung

Ein Event ist ein Ereignis, das entweder erwartet aber zeitlich nicht bekannt aufgrund äußerer Bedingungen auftritt, oder auch zyklisch ausgelöst wird. Wenn beispielsweise ein Wagen in einer Automatisierungsanlage eine Lichtschranke durchfährt, kann für die Steuerung damit ein Event ausgelöst werden. Arbeitet man insgesamt eventgetrieben, dann ist es zweckmäßig zyklische Events zu erzeugen als Ersatz für eine zyklische Abtastzeit.

6.1 Events und Objektorientierung

Events sind zusammen mit der Objektorientierung favorisiert worde, wie folgendes Zitat zeigt, Objektorientierte Programmierung - Wikipedia einleitend findet sich folgendes Zitat von Alan Kay, Miterfinder von Smaltalk und Pionier der Objektorientierung, siehe auch Kapitel 2 Objektorientierte Programmierung, Seite 4:

1. Everything is an object,

2.Objects communicate by sending and receiving messages (in terms of objects),

3.....

6. To eval a program list, control is passed to the first object and the remainder is treated as its message

1. Alles ist ein Objekt,

2.Objekte kommunizieren durch das Senden und Empfangen von Nachrichten (welche aus Objekten bestehen),

3.,,,,

6.Um eine Programmliste auszuführen, wird die Ausführungskontrolle dem ersten Objekt gegeben und das Verbleibende als dessen Nachricht behandelt

(Übersetzung laut Wikepedia, abgerufen in o.g. Link am 2019-07-15)

Folgerichtig ist die eventgetriebene Programmierung Bestandteil der UML als Grafische Ausdrucksmöglichkeit der Objektorientierung. Nicht folgerichtig ist, dass in UML-Sequenzdiagrammen alle Interaktionen zwischen Objekten als "Message" dargestellt wird. Eine Message (Nachricht) ist der Transportmechanismus eines Events. Hier ist offensichtlich der 2. Satz von Alan Kay direkt umgesetzt worden, als Historie der Entstehung des UML. In der praktischen Objektorientierten Programmierung wird viel häufiger eine Operation (Methode) einer anderen Klasseninstanz gerufen als es eine Eventkopplung gibt.

Einfach zusammengefasst: Objektorientierung und Eventkopplung haben an sich nichts miteinander zu tun. Man kann sie aber zweckmäßig koppeln.

6.2 Prinzip Message und Event

Die Message ist die Übertragung eines Event. Das Event selbst wird generiert aus einem Softwareteil. Das kann beispielsweise eine zyklische (oder eventzyklische) Abarbeitung sein, die aus der Abfrage eines Eingangssignales in einer if-Verzweigung die Message sendet. Es kann aber auch ein Interrupt sein, der im Prozessor unmittelbar hardwareverdrahtet aufgerufen wird und die Message sendet. Der Empfänger der Message generiert dann dort wieder ein Event, dass die Message verarbeitet. Die Benutzung der Begriffe Event und Message ist häufig nicht ganz sauber, man darf nicht den Begriff als Eigenname nutzen sondern muss dessen Bedeutung beachten. So kann es keine Event-Queue geben, Events sind Ereignisse und können nicht gespeichert werden. Es handelt sich um die Message-Queue.

Das Senden und Empfangen von Messages ist Interprozesskommunikation, Kommunikation zwischen Geräten oder einfach nur der Aufruf einer Write-Operation, um die Message in die Empfangs-Queue einzuspeichern. Das Medium der Übertragung hat nichts mit dem Prinzip zu tun. Mehr noch, das Medium kann ausgetauscht werden ohne die Anwendersoftware ändern zu müssen. So kann eine einheitliche Anwendersoftware beispielsweise auf verschiedenen Plattformen laufen.

Beim Senden und Empfangen spielen Queues eine entscheidende Rolle (Fifo-Buffer). Die Queues oder Buffer sorgen für die zeitliche Entkopplung. Es können beispielsweise in einem Abtastschritt sehr viele Messages erzeugt werden, die dann erst nacheinander gesendet und abgearbeitet werden können. Dabei spielt beim Senden die Queue nur eine organisatorische Rolle, wesentlicher ist die **Empfangs-Queue und deren Abarbeitungsprinzipien**:

Der Empfang von Messages in der Kommunikations-Zwischenebene erfolgt in einem Empfangsthread (oder Interrupt), der ggf. den Verarbeitern zugeordnete mehrere Queues beschreibt.

Eine Queue muss genau einem Verarbeitungsthread zugeordnet sein. In diesem Verarbeitungsthread werden die Messages zunächst nicht applikationsbezogen einzeln nacheinander aus der Queue gelesen und dann (aufgrund einer Empfängerinformation in der Message) der betreffenden Instanz zur Auswertung übergeben. Erst wenn die Message als Event vollständig ausgewertet ist (häufig für das Schalten von Transitions in Statemachines), wird das nächste Event abgeholt. Diese Sequentialisierung sollte auch eingehalten werden, wenn die Messages unabhängig (wirklich ?) in verschiedenen Instanzen ausgewertet werden. Will man parallelisieren, dann bedarf es mehrerer Threads und mehrerer Empfangsqueues. Gibt es hierbei ein Synchronisationsproblem, muss das dann auf der Anwenderebene gelöst werden.

Die Verarbeitung einer Message als Event bis zur vollständigen Ausführung wird bei Statemachines als "*Run-to-completion*"-Zyklus bezeichnet (Siehe /omg-UML/.Abschnitt *14.2.3.9.1 The run-to-completion paradigm* S. 314. Wichtig dabei ist, dass auch alle folgenden nur durch Bedingungen bestimmte Stateübergänge ausgeführt werden, dabei erzeugte Events zunächst in der (Sende-) Queue landen und deren Ergebnis eben nicht sofort wirksam ist.

6.3 Events in der UML

Für gewöhnliche Programmierung von Embedded Controllern in C/++ oder adäquaten Sprachen (Java, C#) wird häufig UML verwendet. Unabhängig vom Thema Objektorientierung steht damit pro class ein Statechart zur Verfügung, der beliebig komplex (nested, parallel) sein kann. Man erhält damit eine "Verhaltensbeschreibung" ("*Behaviour*") und eine entsprechende eventgesteuerte Abarbeitung.

Die Statecharts in UML lassen sich allerdings auch rein bedingungsgesteuert abarbeiten, in diesem Fall ohne Events. Ein häufiges Argument ist dabei: Für eine schnelle Abarbeitungszeit ist eine Eventsteuerung weniger zu gebrauchen. Ob das Argument immer richtig ist, sei dahingestellt.

6.4 Events in der Programmierung Grafischer Benutzeroberflächen

In Grafischen Benutzeroberflächen sind Events seit der Entwicklung von Widget-Systemen präsent. Das Event selbst ist die Bediehandlung des Anwenders. Zu bestimmten Bedienhandlungen werden Operationen registriert, die dann als Eventhandler bezeichnet werden. Diese Anwendung der Events ist den meisten PC-Programmierern als "Eventverarbeitung" geläufig.

6.5 Events in der Objektorientierung

Hier wäre zu wiederholen, was einleitend im Abschnitt 2 Objektorientierte Programmierung, Seite 2 benannt wurde. In der Objektorientierung ist Eventhandling mit dem Messaging mit bestimmten in verbreiteten Libraries aber nicht standardisiert vorhandenen Mechanismen zu realisieren. QT hat mit der Signal-Slot-Funktionalität seinen eigenen QT-Standard geschaffen.

6.6 Events in der Automatisierungstechnik, IEC 61499

In der Automatisierungstechnik ist der Hype der Eventverarbeitung mit UML und Objektorientierung in den 1990-ger Jahren zunächst vorbeigegangen. Das ist insoweit nicht unbedingt folgerichtig, da in der Automatisierung die Reaktion auf äußere Ereignisse an der Tagesordnung ist. Man hatte jedoch eine Norm IEC 61131, aus den Ende-70-gern mit einer eher statischen Herangehensweise. Typische Denkmuster sind die Flankentriggerung aus einem binären Eingangssignal und deren Verarbeitung mit Freigabesteuerung einzelner Funktionsblöcke über EN und ENO, was eher einer bedingten Ausführung entspricht.

Wenn mehrere Automatisierungsgeräte zusammenarbeiten wollen, dann wurde nach der IEC 61131-Norm arbeitend spezifische Kommunikationskanäle für logische oder numerische Signale entworfen.

Das hat sich mit der Norm **IEC 61499** entstanden ab ca. 2006, die auf der gängigen IEC 61131 aufbaut, wesentlich geändert. Die Grundidee dieser Norm ist die **Verteilte Automatisierungstechnik** mit Ausblendung des Kommunikationsaspektes auf der Anwenderebene der Automatisierungs-programmierung. Es ist schlichtweg egal, auf welchem Gerät ein Binärsignal ankommt (dies ist mit dem Anlagenaufbau determiniert). Es wird ein gemeinsames Programm der Automatisierung erstellt, mit Funktionsblöcken, die erst im zweiten Schritt den einzelnen Automatisierungsgeräten aufgrund der Hardwareverhältnisse zugeordnet werden. Man braucht bei einer Änderung der Hardwareverhältnisse (Binärsignal nicht mehr am Automatisierungsgerät X234 sondern an X345 angeschlossen) keine Kommunikationsverbindungen zu korrigieren und keinen Funktionsplan zu ändern. Lediglich die FBlock-Zuordnung zu den Geräten ("*Devices*", und feingranularer "*Ressources*" innerhalb des Device als einzelner Ablaufcode) muss in der *SystemConfiguration* angepasst werden, was eine einfache Parametrierungsaufgabe ist.

Die Funktionsblöcke sind generell mit Events verbunden. Damit ist es egal, ob aufeinanderfolgende FBlocks sich in der selben Ressource befinden und damit defakto einfach nacheinander abgearbeitet werden, oder ob die zugehörigen Austauschdaten per Eventkopplung über die Kommunikation übertragen werden, der Aufruf also nach Kommunikation auf dem anderen Device erfolgt. Um die Kommunikation muss sich der User selbst nicht kümmern. Da die Eventkommunikation mit Zuordnung der FBlocks zu Devices und Ressources feststeht, können die notwendigen Kommunikationsdatenstrukturen automatisch richtig erstellt werden. Wichtig ist anzumerken, dass die Datenverbindungen (Ein-/Ausgänge der FBlocks) den Event-Pins zugeordnet sind, folglich werden die Daten mit passend übertragen.

Die Norm IEC 61499 wurde nach einem Anfangshype bisher nicht flächendeckend bei den Automatisierungsgeräteherstellern eingeführt. Das liegt auch daran, dass bereits spezifische Lösungen vorhanden waren und sind, die nicht einfach aufgegeben werden sollten. Der Druck von Kunden in Richtung herstellerunabhängiger Projektierung war bisher offensichtlich nicht genügend groß. Meistenteils hat man seinen gewohnten Lieferanten der Hardware, der das passende genau auf die Hardware abgestimmtes Tooling mit liefert.

Eine Normungsrunde der IEC 61499, die für das Jahr 2020 vorgesehen ist, beschäftigt sich unter anderem mit der konsequenten Verbesserung der Statemachine-Technik, die im wesentlichen kompatibel mit den aus UML bekannten Statecharts sind (nach David Harell).

Eine mögliche Verwendung der IEC61499 als Abbild der grafischen Verdrahtung von Funktionsblöcken und Zwischenformat vor der Embedded-Code-Generierung (FBCL = "*FunctionBlockConnectionLanguage*") ist ebenfalls im Fokus der Entwicklung.

6.7 Events und Abarbeitungsreihenfolge

Die konsequente Eventkopplung von einzelnen Funtionsblöcken in der IEC 61499 zeigt viel stärker als in UML, dass es eine Überführung zwischen Eventkopplung mit Queue (Fifo.Buffer) und der einfachen Abarbeitungsreihenfolgebestimmung gibt. Voraussetzung für den zweiten Fall ist es, dass sich die eventgekoppelten FBlocks als *Chain* in der selben Ablaufumgebung (Ressource) befinden. Für die Kommunikation über Ressourcegrenzen (Threads, Devices) ist die Eventqueue nach wie vor die richtige und einzige Lösung.

Für die Verwendung der Events zur Abarbeitungsreihenfolgebestimmung ist weiterhin eine Eventsynchronisation (Rendezvous-FBlock in IEC 61499) und Parallelverarbeitung aufgrund aufgespalteter Eventchain (Eventverdrahtung) wesentlich.

Ein `E_REND`-Funktionsblock nach IEC 61499 bedeutet, dass zwei oder mehrere Events eingetroffen sein müssen, bevor die am Ausgang angeschlossenen FBlocks abgearbeitet werden. Für reine Eventkopplung braucht es dazu eine Statemachine, die auf die einzelnen Events parallel reagiert und mit einer Join-Transition dann das Outputevent erzeugt. Für die Abarbeitungsreihenfolgebestimmung bedeutet dies, dass alle FBlocks der hinführenden Event-Chain zuerst aufgerufen werden müssen. Danach kommen die FBlocks der Folge-Chain dran.

Ein Event kann in der IEC 61499 an mehrere Ziel-FBlocks verdrahtet sein. Für die Abarbeitungsreihenfolge bedeutet das, dass entweder eine parallele Verarbeitung der Funktionen möglich ist (Multicore-Umgebung) oder diese Chains nacheinander abgearbeitet werden bis zum Output-Event oder bis zur Vereinigung der Events mit einem E_REND. Damit ergibt sich ein Potenzial, Parallelverarbeitung auf der Ebene der Event-Connection für Mulitcore-Anwendungen zu planen.

Ein Event-Input kann parallel verdrahtet von mehreren Event-Outputs (-chains) getriggert werden. Diese Möglichkeit ist nicht als Abarbeitungsreihenfolge umsetzbar, denn die Output-Events werden kaum gleichzeitig kommen. An dieser Stelle muss die Verarbeitung über eine Event-Queue organisiert werden.

Wenn die Events für das Triggern von Statemachines verwendet werden, dann ist das selbstverständlich keine Abarbeitungsreihenfolge sondern eben Statemachine-Technik. Die Abarbeitungsreihenfolge-Thematik gilt nur für rein funktionale FBlocks, die entweder Composite FBlocks sind, oder FBlocks mit interner imperativer Programmierung (bei IEC 61499 in ST, Structure Text). Es ist allerdings auch möglich, FBlocks in IEC 61499 als Interface zu designen und die interne Abarbeitung unabhängig davon direkt in der Zielsprache (zum Beispiel C) bereitzustellen.

6.8 Sind Messagekopplungen mit Eventverarbeitung langsam?

Sie sind langsamer als der direkte Aufruf einer Operation, wenn diese in der gleichen Ressource stehen. Daher wird für die Anwendung der IEC 61499-Norm für Embedded Controll in schneller Echtzeit die Eventkopplung auch in eine Aufrufreihenfolge umgesetzt.

Für typische Zykluszeiten der Automatisierungstechnik oder erwarteten Reaktionszeiten ist die Eventkopplung genügend schnell. In der schnellen Echtzeitverarbeitung von Einzelfunktionen geht es um den Nanosekundenbereich als Laufzeit einer Operation. Das Ausräumen einer Eventqueue und der Aufruf der Verarbeitungsfunktion benötigt ebenfalls nur einige 10 Nanosekunden bei durchschnittlichen Prozessoren. Werden einzelne numerische Befehle grundsätzlich über Eventkopplung aufgerufen, dann verlangsamt sich die Ausführungsgeschwindigkeit selbstverständlich um Größenordnungen.

Die Eventkopplung ist also geeignet, die Abarbeitung größerer Einheiten zu organisieren. Dabei ist man immer noch im kleinen Mikrosekundenbereich unterwegs. In verteilten Anwendungen muss freilich die Zykluszeit der Kommunikation in Betracht gezogen werden. Damit vergrößern sich Reaktionszeiten in durchschnittlichen Systemen auf den Bereich >100 µs.

6.9 Handshake und Timeout

Eine Message ist eine Information in eine Richtung. Dabei kann nicht vorausgesetzt werden, dass die Message ankommt und richtig verarbeitet wird. In technischen Systemen gibt es Ausfallwahrscheinlichkeiten, Beispiel Stromausfall oder ein herausgefallener Stecker.

Hängt die Verarbeitung von einer Antwortmessage ab, und diese Antwortmessage wird unbedingt erwartet, kann damit ein System hängen, nicht mehr bedien- und debugfähig sein. Folglich muss immer mit dem Ausfall der Antwort gerechnet werden. In Statemachines ist dafür die Timeout-Transition vorgesehen, die also bewusst eingesetzt werden muss. Mit Hierarchischen Statemachines ist es einfach möglich, einen übergeordneten Stateübergang für einen Noteingriff vorzusehen, der unabhängig vom Detailstate einen definierten Zustand wiederherstellt.

Grundsätzlich falsch ist es, etwa in einem Kommunikationsmechanismus Messages aufzuheben und nach einem Fehler-Recovery erneut versuchen zu applizieren. Die verlorene Message darf nicht mit einem allgemeinen System-Mechanismus falsch appliziert werden. Mit Handshake und Timeout ist die Situation auf der Anwenderseite zu klären.

7. Literatur und Links

/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

/mcore/ embedded-software-engineering.de/software-entwicklung-fuer-multicore-systeme

*****