Anwendung der Objektorientierung in einem industriellen Telekommunikationsprojekt

W. Günther, J.F.H. Winkler*

Siemens AG, München, Friedrich-Schiller-Universität, Jena

Zusammenfassung. Der vorliegende Aufsatz berichtet über den Einsatz der Objektorientierung (OO) bei der Entwicklung von SW für Pilot-Vermittlungssysteme. Auch der Einsatz der Objektorientierung hatte in diesem Projekt Pilotcharakter, da es sich um den ersten Einsatz in diesem Bereich innerhalb der Siemens AG handelte. Da es bei den programmiertechnischen Aspekten der OO häufig noch zu Mißverständnissen kommt, werden die Grundkonzepte der OO, so wie sie im vorliegenden Projekt und in Object-CHILL verwendet werden, im ersten Teil noch einmal dargestellt. Im zweiten Teil wird dann über die wesentlichen Charakteristika der im Rahmen des Projektes entwikkelten OO-SW berichtet. Dabei liegt der Schwerpunkt auf der Strukturierung, der verwendeten Entwicklungsmethodik und den verwendeten Werkzeugen. Eine Zusammenfassung der gewonnen Erfahrungen beschließt den Aufsatz.

Schlüsselwörter: Objektorientierung, SW-Entwicklung, SW-Architektur, objektorientierte Programmierung, Object-CHILL

Summary. In the present paper we report about the development of object-oriented SW for a telecom system. The project was a pilot project for the application of object orientation because it was the first application of object orientation in this part of Siemens. In the first part we describe the main concepts of object orientation from a programming language point of view. We do this because there is still some confusion in the SW community about the nature of these concepts. In the second part we describe the main features of the OO SW developed in this project. We report especially about the development methodology and tools used, and about the architecture of the SW. The paper ends with some remarks about the experiences gained.

Key words: Object-oriented techniques, SW development, SW architecture, object-oriented programming, Object CHILL

Computing Reviews Classification: D.1.5, D.2.2, D.2.9, D.3.2

* Während der Arbeiten, über die hier berichtet wird, war J. Winkler Mitarbeiter der Siemens AG, München.

1. Einleitung

Die OO wurde "erfunden" im Gebiet der Programmiersprachen [8; 13; 22] und stellt dort eine nützliche Fortsetzung einer Entwicklung dar, die von der prozedurorientierten über die modulare zur objektorientierten Programmierung führte. In diesem Bereich ist die OO auch relativ präzise definiert. Eine häufig zitierte Charakterisierung ist die von Wegner [28]: OO = objects + classes + inheritance. Diese hat allerdings mit dazu beigetragen, daß Vererbung als essentiell für die Objektorientierung angesehen wird, obwohl diese Sicht durch die Genese der OOP keineswegs gedeckt ist [22]. Dabei bedeutet Vererbung hier näherungsweise "Inkrementelle Erweiterung ohne Modifikation". Wirth hat dafür die Kurzformel "Type Extensions" geprägt [33].

Mit einer gewissen Zeitverzögerung haben auch andere Kreise der Software-Technik (SWT) die OO aufgegriffen und in anderen Phasen der SW-Entwicklung angewendet wie z.B. Requirementsanalyse oder Design. Hier wird die OO oft anders gesehen und charakterisiert, und zwar wird die Vererbung fast immer als eine Realisierung der begrifflichen Klassifikation dargestellt [32]. Diese unterschiedlichen Sichten führen häufig zur Verwirrung ["Guaranteed to confuse": 25: 39; 32]. Aus diesem Grunde werden die Grundkonzepte der OO, so wie sie im vorliegenden Projekt und in Object-CHILL verwendet werden, im ersten Teil noch einmal dargestellt.

Im zweiten Teil des Aufsatzes wird über die Anwendung der OO in einem mittelgroßen industriellen Projekt aus dem Bereich der Telekommunikation berichtet und über die Erfahrungen, die dabei gemacht wurden. Diese Erfahrungen sind positiv, dämpfen aber auf der anderen Seite die etwas zu enthusiastischen Erwartungen, die häufig im Zusammenhang mit der OO geäußert wurden und werden. Durch die gewonnene Erfahrung wird die OO in ein realistischeres Licht gerückt. Die guten Eigenschaften bezüglich der Programmstrukturierung haben sich bestätigt. An der Einteilung des SW-Entwicklungsprozesses in die Hauptschritte Anforderungsanalyse, Grob- und Feinentwurf, Implementierung, Test, Einsatz mußte nichts geändert werden, innerhalb der einzelnen Schritte ergaben sich unterschiedliche Einflüsse der OO. Die OO wirkt sich hauptsächlich so aus, daß das Ausdrucksrepertoire des SW-

Ingenieurs um die Konzepte der OO erweitert wird. Im vorliegenden Fall bedeutet dies, daß die bisher verwendete Programmiersprache CHILL [16; 19; 26; 30] um die Elemente der OO erweitert wurde. CHILL bietet bereits Einzelobjekte in Form von Moduln und unterstützt daher bereits Modularisierung, Kapselung und Datenabstraktion. Die Hinzufügung von Objekttyp, Vererbung und Generizität bewirkt eine Erweiterung des Ausdrucksrepertoires für die Strukturierung von SW. Eine Beschränkung nur auf die Ausdrucksmittel der OO, wie sie häufig postuliert und gefordert wird, erscheint technisch nicht sinnvoll. Wenn ein Programmbaustein konzeptionell eine Funktion oder ein Modul ist, dann ist es nicht hilfreich, ihn künstlich und "mit Gewalt" in einen Objekttyp zu pressen. Der erfahrene Fachmann verwendet jeweils die Konstruktion, die für das aktuelle Problem am besten geeignet ist, und beschränkt sich nicht künstlich auf ein Teilrepertoire der zur Verfügung stehenden Hilfsmittel.
Außer den Konzepten der OO wurden den SW-Ingenieuren noch weitere neue Konzepte zur Verfügung gestellt (generische Bausteine), und eine erste Untersuchung des Einsatzes der neuen Konzepte ergab, daß Vererbung und Generizität etwa gleichhäufig eingesetzt wurden. Daß beide Konzepte, Vererbung und Generizität, gebraucht werden, wurde bereits von Meyer festgestellt [23].
Zu beachten ist hierbei, daß im Umfeld des Projektes seit etwa 15 Jahren CHILL eingesetzt wird, das bereits Elemente zum Programmieren im Großen enthält (s.o.). D.h. im Umfeld des Projektes wird bereits seit längerer Zeit bis zu einem gewissen Grade "in Objekten gedacht", wenn auch ohne den Einsatz der Vererbung. Daher spielen auch die heute in der Diskussion über SW-Strukturen häufig genannten Strukturierten Techniken (SA, SD) hier keine Rolle und müssen insbesondere auch nicht überwunden werden. In einer solchen Welt ist die Einbeziehung der OO ein weniger dramatischer Schritt als z.B. in einer Welt des klassischen Pascal, welches zur Gliederung nur Prozeduren bietet.
Manchem Leser werden unsere Äußerungen zu den Erfahrungen vielleicht zu allgemein und unverbindlich sein, er möchte gern mehr "harte Zahlen und Fakten". Dazu möchten wir anmerken, daß es verfrüht wäre, aufgrund *eines* Projektes zu behaupten, die dabei gemachten Erfahrungen seien allgemeingültig. Solche allgemeingültigen Erfahrungen werden sich nur aus einer Vielzahl von Projekten herauskristallisieren. Da es bisher aber nur wenig Berichte über die Anwendung der OO in größeren Projekten gibt, halten wir die dabei gemachten Erfahrungen für sehr nützlich und möchten andere Anwender zu ähnlichen Veröffentlichungen ermuntern, um die Diskussion über die OO in eine etwas mehr an der Praxis orientierte Richtung zu bringen.

2. Grundkonzepte der OO

Als Grundkonzepte der OO werden oft genannt:

Objekt
Klasse / Objekttyp
Vererbung
Polymorphismus

2.1. Objekt

Ein Objekt ist ein Programmbaustein, welcher in seiner Externschnittstelle Prozeduren, Datentypen, Konstante und manchmal auch Variable anbietet. Technisch besteht eine große Ähnlichkeit zu Ada-Paketen oder Modula-Moduln. Intern können weitere Größen wie Variable, Typen usw. definiert sein. Für manche OO-Programmiersprachen wie Turbo-Pascal, Oberon oder die für Ada vorgeschlagene Erweiterung gilt dies nur mit Einschränkungen: alle drei genannten Sprachen erlauben es z.B. nicht, Konstante oder Datentypen in einem Objekttyp zu definieren [5: 35, 37; 18: 8; 24: 287]. Der Grund dafür ist, daß in allen drei Sprachen die Objekttypen von den Verbundtypen abgeleitet werden und nicht von Paketen oder Moduln. Der letztere Weg wurde in Object-CHILL beschritten.
Die Entwicklung der Software-Technik (SWT) der vergangenen zwei Jahrzehnte hat gezeigt, daß der typische Baustein von SW-Systemen oder sogar DV-Systemen eine Kombination von Daten und Prozeduren (Operationen) ist und nicht die einzelne Funktion (Prozedur, Operation), welche in der Anfangszeit der Programmierung die Programmstruktur dominiert hat. Die ersten Programmiersprachen waren in diesem Sinne prozedurorientiert (Fortran, Algol, PL/1, Pascal), während die nächste Generation Objekte im obigen Sinne als Programmbaustein anbieten (Ada, CHILL, Modula, Turbo-Pascal).
Im Gegensatz zu den Paket- und Modul-Objekten der letzteren Sprachen sind die Objekte in den neuen OO-Sprachen als Variable einer neuen Typart realisiert, der am konsequentesten in den OO-Pascal-Dialekten als Objekttyp bezeichnet wird. Häufig werden diese Typen auch als Klassen bezeichnet, und dieser Begriff wird auch sonst viel verwendet, obwohl er zu Mißverständnissen Anlaß geben kann [32].
Beispiele für Objektdeklarationen sind (die verwendete Sprache ist Object-CHILL [9; 31]):

```
DCL StapelA, StapelB  Stack;    /*
       Statische Objekte */
```

In dieser Deklaration werden zwei Objekte mit den Namen `StapelA` und `StapelB` deklariert, deren Typ `Stack` ist. Als Beispiel dient hier der oft verwendete Stapelspeicher. Die Gründe hierfür sind: das Beispiel ist klein genug und ist inhaltlich weitgehend bekannt.
Wie in den typischen prozeduralen Sprachen üblich können auch dynamische Variable, deren Typ ein Objekttyp ist, erzeugt werden:

```
DCL  VerweisAufStapel  REF Stack;
VerweisAufStapel := NEW Stack;        /*
Dyn. Objekt */
```

Die Struktur und das Verhalten der durch die Deklaration bzw. Operation `NEW` erzeugten Objekte ist durch den Typ `Stack` bestimmt, der im nächsten Abschnitt dargestellt ist.

2.2. Klasse / Objekttyp

Wie bereits im Abschnitt 2.1 erwähnt ist der Begriff "Objekttyp", der auch in den OO-Pascal-Dialekten verwendet wird, besser als der Begriff "Klasse" geeignet, Mißverständnisse zu vermeiden. Daher wird in diesem Aufsatz durchweg "Objekttyp" verwendet.

Eine Definition des Typs `Stack` kann folgendermaßen aussehen:

```
Stack: MODULE CLASS
   GRANT Push, Pop PREFIXED Stack;   /*
Exportklausel */
   Push: PROC(Value Int);
   Pop:  PROC() RETURNS(Int);
/****  Anfang des internen Teils  *****/
   SYN Length = 100;
   DCL StackData ARRAY(1 : Length) Int;
   DCL TopOfStack RANGE(0 : Length) INIT
:= 0;
END Stack;
```

Dies ist die Spezifikation des Objekttyps. Diese Spezifikation enthält die Definition der Externschnittstelle. Aus implementierungstechnischen Gründen enthält die Spezifikation auch interne Größen ähnlich wie der Private-Teil eines Ada-Paketes [2: 8-11]. Aus der Typ-Spezifikation geht hervor, daß ein `Stack`-Objekt in der Extern-Schnittstelle die zwei bekannten Operationen `Push` und `Pop` enthält. Diese Operationen werden als Prozeduren realisiert. Oft werden solche Prozeduren auch als Methoden bezeichnet. Intern ist der Stapel als eine Reihung (Array) realisiert. Überlauf und Unterlauf werden in diesem kleinen Beipiel nicht als Fehler abgefangen.
Zur Definition eines Objekttyps gehört auch ein Rumpf, in welchem weitere interne Größen und insbesondere die Rümpfe der in der Spezifikation definierten Prozeduren enthalten sind.

```
Stack: MODULE BODY
   . . .
   Pop:  PROC() RETURNS(Int);
      IF  TopOfStack > 0
      THEN RESULT StackData(TopOfStack);
           TopOfStack := TopOfStack-1;
      FI;
   END Pop;
   . . .
END Stack;
```

Aus Platzgründen ist nur der Rumpf der Prozedur `Pop` ausgeführt.
Die Manipulation von Objekten, d.h. Variablen deren Typ ein Objekttyp ist, erfolgt dadurch, daß eine exportierte Prozedur auf das Objekt angewendet wird:

```
StapelA.Push(10);
VerweisAufStapel->.Push(20);
```

Wie man am vorliegenden Beispiel sieht, unterstützen die Objekttypen Kapselung und Information Hiding, wie sie in den letzten 20 Jahren diskutiert werden. Objekttypen in Object-CHILL eignen sich daher gut zur Realisierung von Datenabstraktionen.

2.3. Vererbung

Die Vererbung[1] erlaubt, es Varianten eines existierenden Objekttyps T zu erstellen, ohne T zu modifizieren und ohne die in T vorhandenen Komponenten (Daten, Prozeduren) zu kopieren. Es ist damit eine inkrementelle Entwicklung möglich.
Die Vererbung unterstützt insbesondere die Wiederverwendung, und zwar eine Wiederverwendung durch Bezugnahme und nicht durch Kopieren.
Wenn ein Stack-Typ benötigt wird, der auch noch eine Operation `Top` anbietet, die nur das oberste Element zurückliefert, ohne das Stackobjekt selbst zu verändern, dann kann dies dadurch erfolgen, daß ein Objekttyp `StackMiTop` definiert wird, der nur diese neue Prozedur enthält und ansonsten auf den bereits vorhandenen Typ `Stack` Bezug nimmt:

```
StackMitTop:  MODULE CLASS  IS_A Stack
                  /* Vererbungsklausel */
   GRANT  Pop,  Top PREFIXED Stack2;
   Top:  PROC() RETURNS(Int);
END StackMitTop;

StackMitTop:  MODULE BODY
   Top:  PROC() RETURNS(Int);/* Neue Proz. */
      IF  TopOfStack > 0
      THEN RESULT StackData(TopOfStack);
      FI;
   END Top;
END StackMitTop;
```

Die Wirkung der Vererbungsklausel "`IS_A Stack`" ist nun so, daß sich logisch gesehen `StackMitTop` so verhält, als wären alle Größen, die in `Stack` deklariert sind, auch in `StackMitTop` deklariert, während physisch gesehen, der Programmtext von `StackMitTop` nur das funktionelle Delta zwischen `Stack` und `StackMitTop` enthält. Damit ist nun folgendes möglich:

```
DCL  StapelC StackMitTop;
DCL  I  Int;
StapelC.Push(10); /* in Stack definiert */
I := StapelC.Top();/* in StackMitTop definiert *
```

Im Beispiel werden die Operationen `Push` und `Top` völlig gleichartig auf `StapelC` angewendet, obwohl `Push` in `Stack` und `Top` in `StackMitTop` definiert ist. Dies ist ein typisches Beispiel für den Einsatz der Vererbung, um eine inkrementelle Erweiterung eines bestehenden Objekttyps zu definieren. `Stack` wird häufig als *Basistyp* oder *Vorfahre* und `StackMitTop` als *abgeleiteter Typ* oder *Nachkomme* bezeichnet.
Die Vererbung kann noch in einer zweiten Art eingesetzt werden. Dabei ist es möglich, im abgeleiteten Typ einen neuen Rumpf für eine im gegebenen Typ vorhandene Prozedur zu formulieren. Man spricht dann von einer Reimplementierung. Dies kann dann eingesetzt werden, wenn für neue Anwendungen der bisherige Rumpf nicht brauchbar ist. Im vorliegenden Beispiel könnte es z.B. nötig sein, daß die Prozedur `Pop` das gerade freigewordene Element des Stapels mit einem definierten Wert (z.B. 0) besetzt. Außerdem sollen auch nach Erzeugung des Objektes alle Stapelkomponenten mit diesem Wert belegt sein. Dies führt zu Definition des Typs `StackMitPopneu`:

```
StackMitPopneu:    MODULE   CLASS    IS_A
Stack
                  /* Vererbungsklausel */
   GRANT  Pop  PREFIXED StackMitPopneu;
   Pop:  PROC() RETURNS(Int);
   StackMitPopneu: CONSTR();
```

[1] "To understand inheritance better, we look first at the role of inheritance in programming languages." [7]

```
END StackMitPopneu;

StackMitPopneu:  MODULE BODY
   SYN  Empty  = 0;      /* Konstante */
   Pop: PROC() RETURNS(Int);         /*
Reimpl. Proz. */
      IF  TopOfStack > 0
      THEN RESULT StackData(TopOfStack);
           StackData(TopOfStack) :=
Empty;
           TopOfStack := TopOfStack-1;
      FI;
   END Pop;

   StackMitPopneu: CONSTR();
     DCL I RANGE(1 : Length);
     DO  FOR  I := 1 TO Length;
         StackData(I) := Empty;
     OD;
   END  StackMitPopneu;
END StackMitPopneu;
```

Der Typ `StackMitPopneu` ist ebenfalls direkt aus dem Typ `Stack` abgeleitet. Er unterscheidet sich aber in der Wirkung der Operation `Pop`. Außerdem enthält `StackMitPopneu` noch einen Konstruktor, der genauso heißt wie der Typ selbst. Dieser Konstruktor wird beim Erzeugen eines `StackMitPopneu`-Objektes automatisch ausgeführt und setzt alle Stapelkomponenten auf den Wert 0.
Im Beispiel von `StackMitPopneu` ist das Inkrement zwischen Basistyp und abgeleitetem Typ noch "kleiner" als im Falle `StackMitTop`, da lediglich der Rumpf einer bereits vorhandenen Prozedur ersetzt wird (außerdem wurde der Konstruktor definiert, der aber nur implizit aufgerufen wird).
Damit sind folgende Manipulationen möglich:

```
DCL  StapelD  StackMitPopneu;
DCL  I  Int;
StapelD.Push(30);    /* Geerbte Proz. */
I := StapelD.Pop();  /* Reimpl. Proz. */
```

Zur Veranschaulichung ist in Abb. 1 die Vererbungshierarchie, die von den drei definierten Objekttypen gebildet wird, dargestellt.

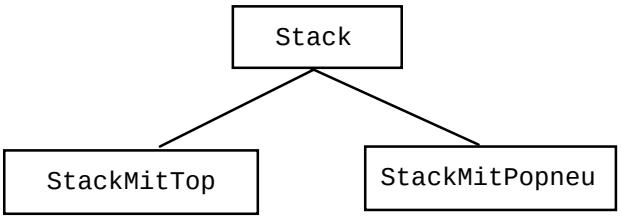

Abb. 1: Vererbungshierarchie

2.4. Polymorphismus

Wie der Abschnitt über Vererbung zeigt, können ein Objekttyp und die von ihm mittels Vererbung abgeleiteten Objekttypen als eine Menge verwandter Varianten gesehen werden analog zu den Verbund-Varianten in Ada oder Pascal. In OO-Sprachen mit strenger Typbindung wären diese Varianten stärker voneinander isoliert als diese Verbundvarianten, da jede Objekttyp-Definition einen eigenen Typ definiert. Um diese Trennung zu überwinden, ist nun die Regel für die Typkompatibilität von Verweisvariablen in den OO-Sprachen mit strenger Typbindung abgeschwächt:

```
DCL  VerweisAufStapel  REF Stack;
    /* VerweisAufStapel darf auf Objekte
des Typs
       Stapel oder
       irgendeines davon mittels
       IS_A abgeleiteten Typs verweisen
*/
IF  Bedingung
THEN VerweisAufStapel := NEW Stack;
ELSE     VerweisAufStapel     :=     NEW
StackMitPopneu;
FI;
VerweisAufStapel->.Push(40);
I := VerweisAufStapel->.Pop();
```

Je nachdem, welcher Zweig der IF-Anweisung durchlaufen wurde, verweist `VerweisAufStapel` auf ein `Stack`-Objekt oder auf ein `StackMitPopneu`-Objekt. Nach dem Dereferenzierungspfeil dürfen natürlich nur Größen aus der Externschnittstelle von `Stack`, dem Basistyp von `VerweisAufStapel`, verwendet werden. Diese Vorschrift garantiert, daß für alle zulässigen Werte von `VerweisAufStapel` die nach dem Derefenzierungspfeil auftretenden Größen auch existieren. Würde man z.B.

```
I := VerweisAufStapel->.Top();
```

zulassen, dann wäre die rechte Seite der Zuweisung nicht definiert, wenn `VerweisAufStapel` auf ein `Stack`-Objekt verweist (Durchlaufen des THEN-Zweiges).
In der Anweisung

```
I := VerweisAufStapel->.Pop();
```

wird nun, je nachdem welche Alternative der Fallunterscheidung ausgeführt wurde, der in `Stack` oder der in `StackMitPopneu` definierte Rumpf von `Pop` ausgeführt. Die allgemeine Regel ist so, daß immer der zum *aktuellen Objekt gehörige Rumpf* einer Prozedur ausgeführt wird.
Die Abschwächung der strengen Typbindung für Verweise auf Objekte führt nun dazu, daß einerseits Verweisvariable auf Objekte verschiedener (aber verwandter) Objekttypen verweisen dürfen, und daß es andererseits für eine Prozedur innerhalb einer Gruppe verwandter Objekttypen mehrere Implementierungen (Rümpfe) geben kann. Diese Eigenschaften werden als Polymorphismus ("Vielgestaltigkeit") bezeichnet. Dabei wird der Begriff sowohl auf die Verweisvariablen als auch auf die Prozeduren angewendet.
In anderen Sprachen wie z.B. Turbo-Pascal oder C++ ist die Situation etwas komplizierter, da dort nicht alle Prozeduren diesem Mechanismus unterliegen, sondern nur besonders gekennzeichnete [4: 143; 5: 39; 11: 196]. Dadurch kann es vorkommen, daß bei einem Prozeduraufruf ein Rumpf ausgeführt wird, der *nicht* zum aktuellen Objekt gehört.

2.5. Zusammenfassung

Wie man sieht, bieten die Sprachelemente der OO neue Möglichkeiten im Bereich der Programm- und Datenstrukturierung, d.h. die Palette der Sprachelemente für solche

Zwecke wird entsprechend erweitert, und entsprechend betrifft die OO nur gewisse Aspekte der Programmierung. Aspekte wie Programmablauf, E/A oder Nebenläufigkeit sind davon weitgehend unberührt.
Bezüglich der frühen Phasen der SW-Entwicklung ist der Einfluß größer, da dort im wesentlichen nur die Art der SW-Bausteine (Funktion, Objekt, Modul, Prozeß) und ihre Beziehungen untereinander eine Rolle spielen.

3. Erfahrungen mit der Objektorientierung

3.1. Historie

Innerhalb der Siemens AG wird die OO u.a. in einem Pilotprojekt im Bereich der Telekommunikationstechnik verwendet [14]. In diesem Projekt wird Software für Pilot-Vermittlungssysteme entwickelt. In einer Vorstudie (1988/89) wurde zuerst anhand eines Designs untersucht, ob die OO in diesem Anwendungsbereich nutzbringend eingesetzt werden kann. Im wesentlichen sollten die folgenden Fragen beantwortet werden:

- Welche Software-Strukturen entstehen bei Anwendung der OO? Sind diese Strukturen überschaubarer als die existierenden?
- Ist mit dem Einsatz der OO eine spürbare Verbesserung der Produktivität der Entwicklung und der Software-Qualität, insbesondere der Wiederverwendbarkeit und der Änderbarkeit zu erwarten?

Das Ergebnis der Vorstudie war positiv und führte einerseits dazu, im eigentlichen Entwicklungsprojekt die OO einzusetzen und andererseits dazu, die Sprache CHILL zu Object-CHILL zu erweitern [9].
Die Analysephase des Pilotprojektes begann im Frühjahr 1990, und Mitte 1993 sind die ersten Versuchsanlagen erfolgreich zum Einsatz gekommen.
Die Sprache Object-CHILL wurde 1988/89 definiert. Die Entwicklung eines Compilers für Object-CHILL wurde im Frühjahr 1990 begonnen und die erste Version des Compilers wurde Mitte 1991 freigegeben [31].

3.2. Umfang des Pilotprojekts

Die OO-SW für den ersten Prototyp des Pilot-Vermittlungssystems besteht aus ca. 350 Objekttypen, die netto insgesamt etwa 150 kloc umfassen. Erstellt wurde diese SW von ca. 40 Personen. Von den 350 Objekttypen gehören 20 zur Kategorie der "Standardtypen" mit einem Gesamtumfang von etwa 3 kloc. Beispiele für diese sind Datenstrukturen wie Liste und Tabelle oder Objekttypen, die ein Gerüst für Applikationen vorgeben wie "Steuerung", "Anreizempfänger" oder "Meldungsbehälter". Die Standardtypen wurden parallel zu den anwendungsspezifischen Objekttypen für das Pilotprojekt neu entwickelt. Das bedeutet insbesondere, daß am Anfang des Projektes keine Bibliothek vordefinierter Objekttypen vorhanden war.

3.3. Aspekte der Software-Architektur

Die OO-SW im Pilotprojekt ist wie die anderer Telekom-Systeme auf der obersten Ebene in verschiedene Subsysteme oder Applikationen aufgeteilt, die miteinander nur über Botschaften kommunizieren. Diese Botschaften werden oft auch als Anreize oder Meldungen bezeichnet (s.a. [15: 331]).
Die Architektur der OO-SW im Pilotprojekt wird durch mehrere Strukturierungsprinzipien geprägt, die teils durch die OO bedingt sind und teils auch in klassischen SW-Architekturen vorkommen.
Das erste dieser Strukturierungsprinzipien ist die **Vererbungsrelation**, welche für die Objektorientierung spezifisch ist. Bei der Vererbungsrelation ergab sich entgegen mancher Erwartungen [12: 29] eine eher flache Struktur des Vererbungsbaumes: die maximale Höhe beträgt 4 und die durchschnittliche Höhe 1.71, wobei eine Wurzel jeweils die Höhe 1 hat (Für die Klassenbibliothek des Smalltalk-80-Systems wird ein entsprechender Durchschnittswert von 2.75 berichtet [10: 143]).
Die **Benutzungsrelation** ist dadurch definiert, daß ein Objekttyp A einen Objekttyp B benutzt, und zwar in der Regel so, daß ein A-Objekt eine Methode bei einem B-Objekt ruft. Das sei an folgendem einfachen Beispiel gezeigt, in dem bei einem geschachtelten Objekt eine Methode gerufen wird.

```
Port: REGION CLASS
   GRANT SeizeConnection,   /* Export */
         ReleaseConnection, ...
         PREFIXED Port;
   SEIZE CLASS CapacityAdmin;/* Import */

   SeizeConnection: PROC(ConnId   Int,
                         Capacity Int,
                         Response Int);
   END SeizeConnection;
    . . .
/***   Anfang des internen Teils   ***/

   DCL Capacity CapacityAdmin;
    . . .
END Port;
```

```
Port: REGION BODY
   SeizeConnection: PROC(ConnId   Int,
                         Capacity Int,
                         Response Int);
    . . .
   Capacity.SeizeCap(ConnId,Capacity,OK);
             /* Methodenaufruf bei
                geschachteltem Objekt */
    . . .
   END SeizeConnection;
   . . .
END Port;
```

```
CapacityAdmin: MODULE CLASS
   GRANT SeizeCap,ReleaseCap,...PREFIXED
CapacityAdmin;

   SeizeCap: PROC(CustomerId Int,
                  Capacity   Int,
                  Response   Int);
   END SeizeCap;
    . . .
END CapacityAdmin;
```

Diese Konstruktion ist typisch für die Benutzung von Objekttypen untereinander. Wie man daran sieht, wird die Struktur eines objektorientierten Programmes nicht allein durch die Vererbungsrelation bestimmt.
Auf einer gröberen Ebene ist die OO-SW des Pilotprojektes schichtenweise gegliedert (s. Abb. 2). Jeder Objekttyp ist einer der vier Schichten zugeordnet. Das **Schichtenmodell** regelt die innerhalb einer Schicht wie auch zwischen den Schichten erlaubten Benutzungsrelationen. Es handelt sich dabei um ein klassisches Schichtenmodell, in welchem Benutzungen nur "horizontal" und von "oben" nach "unten" möglich sind. Beispielsweise dürfen Ressourcen-Verwalter nur Benutzungsrelationen zu anderen Ressourcen-Verwaltern oder zu Ressourcen haben. Controller einer Applikation dürfen nicht Controller einer anderen Applikation benutzen, sondern müssen sich an den Trigger der anderen Applikation wenden. Die Trigger nehmen alle Anreize, die von einer Applikation verarbeitet werden müssen, vom Betriebssystem entgegen. Derartige Anreize sind beispielsweise Steuerungsnachrichten von der Peripherie, die an dem Pilot-Vermittlungssystem angeschlossen ist, oder die Benachrichtigung über den Ablauf eines Zeitauftrags, der von einer Applikation beim Betriebssystem gestartet wurde.

Jede Applikation der OO-SW (Vermittlungs-, Administrations-, Sicherungs-, Online-Fehlerdiagnose- und -korrektur-SW) ist nach den Regeln dieses Schichtenmodells aufgebaut.

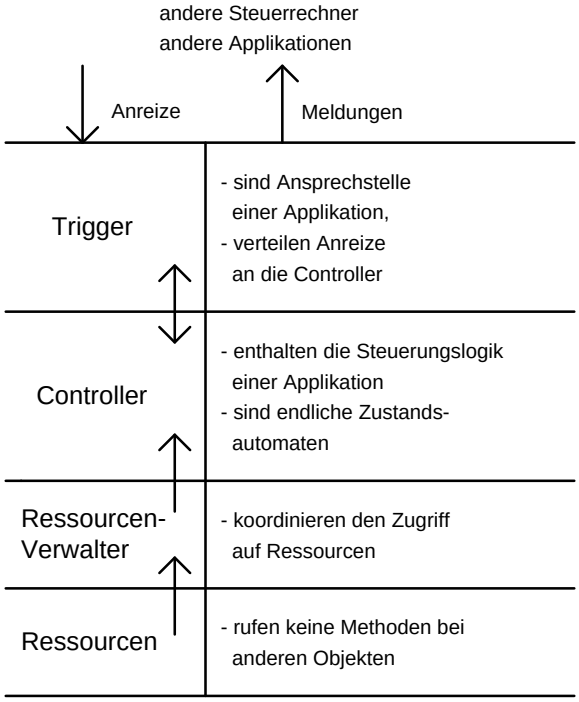

Abb. 2: Schichtenstruktur der OO-SW

Das letzte Strukturierungsprinzip ist die **Ausprägungsrelation**, welche durch die Beziehung von generischen Objekttypen (Schablonen) zu den daraus abgeleiteten nichtgenerischen Objekttypen definiert ist.
Sowohl die Vererbung als auch die generische Ausprägung erlauben es, einen Objekttyp an unterschiedliche Einsatzfälle anzupassen, ohne ihn selbst modifizieren zu müssen. Im Pilotprojekt wurden die Vererbung und die generische Ausprägung zu etwa gleichen Teilen eingesetzt. Als Maß wird dabei die Anzahl der in den Basistypen bzw. generischen Typen enthaltenen Programmzeilen verwendet.
Methoden und Objekte sind im Mittel kleiner als Prozeduren und Module in funktionsorientierten Architekturen. Module enthalten häufig mehrere Datentypen, die bei OO-SW jeweils durch einen eigenen Objekttyp beschrieben werden.
Die drei Relationen eröffnen unterschiedliche Sichtweisen auf die Struktur der OO-SW. Zwischen diesen Sichtweisen scheint zunächst keinerlei Zusammenhang zu bestehen. Eine genauere Analyse der Struktur hat jedoch ergeben, daß es fast keine Vererbungs- wie auch Ausprägungsrelationen über Schichtgrenzen hinweg gibt. Eine Ausnahme stellen die Ressourcen-Verwalter und Ressourcen dar. Hier kommt es vor, daß sowohl Ressourcen-Verwalter als auch Ressourcen in einer Vererbungshierarchie auftauchen.
Zusammenfassend gilt, daß die Struktur der Anwendungs-SW im Pilotprojekt durch folgende Strukturierungsprinzipien gegliedert wird:

Vererbung
Benutzung
Schichtung

Ausprägung

Daß die in der Objetkorientierung bisher übliche Anordnung der Objekttypen auf einer Ebene (in einem Gültigkeitsbereich) für größere Programme nicht ausreichend ist, wird auch von anderen Autoren festgestellt [1; 27; 29].

3.4. Projektmanagement-Aspekte

Es hat sich gezeigt, daß die OO den SW-Entwicklungsprozeß in seiner Grundstruktur weniger beeinflußt als ursprünglich gedacht. Es bleibt bei den bisherigen Phasen

Anforderungsanalyse,
Grobentwurf,
Feinentwurf,
Implementierung und
Test.

Die Anforderungsanalyse umfaßt hier Tätigkeiten wie das Spezifizieren der Anforderungen an das Gesamtsystem sowie der Leistungsmerkmale, die das System aufgrund der Anforderungen besitzen muß. Weiterhin wird festgelegt, welche Hardware- und Software-Subsysteme welchen Anteil an der Realisierung der Leistungsmerkmale haben sollen (Funktionsspezifikation). Zusätzlich wurde für das Software-Subsystem der OO-SW die Aufteilung auf die bereits in 3.3 genannten Applikationen vorgenommen, um den einzelnen Entwicklergruppen sinnvolle Arbeitspakete zuordnen zu können. Die Analysephase für das gesamte HW-SW-System blieb im wesentlichen von der OO unberührt.
Den größten Einfluß hatte die OO auf die Entwurfsphasen für die OO-SW, die im Detail folgendermaßen gegliedert sind:

Grobentwurf
  Objekte identifizieren
  Operationen und Attribute festlegen
  Schnittstellen festlegen
  Objekttypen festlegen
  Vererbungsbeziehungen identifizieren
Feinentwurf
  Internen Aufbau der Objekttypen festlegen
    (d.h. im wesentlichen die Operationen)

Entscheidend für die Arbeiten in der Entwurfsphase war die Einrichtung einer zentralen Koordinationsgruppe ("OOP-Koordinationsrunde") aus 5 - 8 Personen. Eine der Aufgaben der Koordinationsgruppe war die Begutachtung der von den Entwicklungsteams entworfenen Architekturen für die einzelnen Applikationen. Dabei kam es auch vor, daß Entwürfe zurückgewiesen wurden und neue vorgelegt werden mußten. Ebenso hat die Koordinationsgruppe festgelegt, welche Objekttypen in die Gruppe der Standardtypen aufgenommen wurden. Die Mitglieder dieser Gruppe waren einerseits Teilprojektleiter und andererseits die Personen, welche bereits in die Vorstudie involviert waren und daher über relativ viel Erfahrung im Bereich der OO verfügten.

3.5. Einsatz von Formalismen und Sprachen

Bezüglich des Einsatzes von verschiedenen Darstellungshilfsmitteln in den verschiedenen Phasen der SW-Entwicklung ergab sich keine große Veränderung zum bisherigen Vorgehen:

Anforderungsanalyse:
Das Ergebnis der Anforderungsanalyse wird wie bisher überwiegend verbal beschrieben. Zur Darstellung der Informationsflüsse zwischen den verschiedenen Steuerrechnern werden SDL-Blockdiagramme [17; 34] verwendet (SDL = Specification and Description Language). Die Blöcke in diesen Blockdiagrammen haben "Objektcharakter" [6: 113, 114], so daß auch in dieser Phase bereits objektorientiert gearbeitet wurde, auch wenn dieser Begriff im Namen der verwendeten Methode nicht auftritt.
Der zeitliche Ablauf der Informationsflüsse wird mit Message Sequence Charts (MSC) [35] sichtbar gemacht. Diese Diagramme werden auch als Interaktionsdiagramme bezeichnet [6: 142, 143].

Grobentwurf:
Hier werden Kästchen-Kanten-Diagramme eingesetzt, wobei manche dieser Diagramme die Vererbungsrelation darstellen. Andere zeigen die Benutzungsbeziehungen. Die Ablaufstruktur wird mit MSC dargestellt.
Der grobe Aufbau der Objekttypen selbst wird mit Hilfe der Programmiersprache in Form von Spezifikationen realisiert. In diesem Stadium sind die Spezifikationen noch insofern unvollständig, als bei Datenkomponenten nur der Name und noch nicht der Typ angegeben wird. Ansonsten entsprechen diese Spezifikationen aber der Sprache Object-CHILL. Diese Tatsache erleichtert den Übergang in die Implementierungsphase wesentlich.

Feinentwurf:
Im Bereich der Controller-Objekttypen wird die grafische Sprache OSDL (Object-oriented SDL) [3; 34] eingesetzt. Die OSDL-Diagramme werden um kurze Object-CHILL-Fragmente ergänzt und von dem OSDL-Tool in kompilierbare Object-CHILL-Programme umgesetzt. Änderungen an Controller-Objekttypen dürfen nur auf der Ebene der OSDL-Diagramme und nicht im erzeugten Object-CHILL-Programm vorgenommen werden. Für alle anderen Objekttypen werden Nassi-Shneiderman-Diagramme aus Pseudocode erzeugt. Der Pseudocode ist Bestandteil der Programmquelle und ist mit dem effektiven Programmcode auf Stand zu halten. Sowohl bei den Controller-Objekttypen als auch bei den anderen Objekttypen kann somit jederzeit eine aktuelle Programmdokumentation in grafischer Form erzeugt werden.

Implementierung:
Die Implementierung erfolgte in Object-CHILL.

Was auffällt ist der relativ frühe Einsatz einer programmiersprachlichen Notation. Vom technischen Standpunkt ist dies nicht verwunderlich, da die Programmiersprachen vor etwa 10 Jahren mit den Paketen und Modulen von Ada und CHILL Sprachelemente eingeführt haben, die auch zum Einsatz in der Entwurfsphase geeignet sind. Dies scheint aber bisher im Bereich der Software-Technik noch zu wenig zur Kenntnis genommen worden zu sein.

3.6. Einsatz von Werkzeugen

Ein Großteil der vorhandenen Werkzeuge konnte für die OO-SW im Pilotprojekt unverändert weiterverwendet werden (z. B. Dokumentationssystem, Konfigurationsverwaltungssystem, Werkzeuge zur Änderungssteuerung). Andere Werkzeuge mußten erweitert werden wie der CHILL-Compiler, der Debugger und das OSDL-Werkzeug. Einige wenige Werkzeuge wurden für die OO-SW neu entwikkelt. Dazu gehören das Verwaltungssystem (CLASSLIB [21]) und das OO-Testsystem [20].
Die CLASSLIB verwaltet alle Objekttypen in Bibliotheken. Die angebotenen Funktionen umfassen Eintragen und Löschen, Anzeigen des Programmtextes einzelner Objekttypen und Anzeigen von Beziehungen zwischen den Objekttypen. Dabei können Vererbungs-, Benutzungs- und Ausprägungsrelation visualisiert werden. Eine zweite Gruppe von Funktionen dient dem gezielten Finden von Objekttypen nach inhaltlichen Gesichtspunkten zum Zwecke der Wiederverwendung (Retrieval). Hierzu werden die Objekttypen mit Schlüsselworten klassifiziert ähnlich wie bei der Indexierung von Dokumenten. Die Suche geschieht durch Suchanfragen, in welchen ebenfalls Schlüsselworte verwendet werden.

3.7. Ausbildungs-Aspekte

Im OO-Pilotprojekt wurde die Ausbildung in zwei Stufen durchgeführt:

- Ein erster Kurs, welcher in die Konzepte der Objektorientierung einführte.
- Ein zweiter Kurs, in welchem die Programmiersprache Object-CHILL in Theorie und Praxis behandelt wurde.

Während des Projektes wurde das weitere "training on the job" wesentlich durch die Tätigkeit der zentralen Koordinationsgruppe gefördert, da dort im wesentlichen alle Fragen von Methodik und Entwicklungsprozeß diskutiert und gelöst wurden. Die dort gefundenen Lösungen wurden über Protokolle und ein Projekthandbuch allen Mitarbeitern im Projekt bekannt gemacht. Dies hat wesentlich zur Weiterbildung der Projektmitarbeiter beigetragen, da Teile der Vorgehensmethodik aufgrund der Umstände (erstmaliger Einsatz einer neuen Methodik) erst in Verlaufe des Projektes entwickelt werden konnten.

4. Zusammenfassung und Ausblick

Die Ergebnisse und Erfahrungen des Einsatzes der OO in einem mittelgroßen, industriellen Projekt lassen sich wie folgt zusammenfassen:

- OO ist auch für Vermittlungssysteme gut geeignet.
- Die guten Eigenschaften bezüglich der Programmstrukturierung haben sich bestätigt.
- Der Einfluß auf den SW-Entwicklungsprozeß erscheint nicht groß: es bleibt bei den wesentlichen Schritten (Anforderungsanalyse, Grob- und Feinentwurf, Implementierung, Test, Einsatz).
- Bei der Auswertung der Vererbungs- und Ausprägungsrelationen hat sich gezeigt, daß die durch beide Mechanismen bewirkte Codeersparnis je ca. 9% beträgt. Es ist sehr schwierig, hieraus einen Schluß auf den Produktivitätsfortschritt zu ziehen. Dies wäre nur dann möglich, wenn eine vergleichbare Systemimplementierung in konventioneller Technik durchgeführt werden würde. Das kann sich aber bei Projekten dieser Größenordnung niemand leisten. Ein Vergleich von ausgewählten Codierbeispielen hat eine Reduktion des Codes bei der OO-SW im Bereich von 17 - 25%, gemessen in netto Lines of Code, ergeben.
- Es traten keine Probleme mit der dynamischen Effizienz der OO-SW auf.
- Die Einrichtung der zentralen Begutachtungsgruppe hat sich als sehr nützlich erwiesen.
- Ein Großteil der vorhandenen Werkzeuge konnte weiterverwendet werden, einige wurden erweitert.
- Über die Änderbarkeit und Wiederverwendbarkeit der entwickelten SW kann zum jetzigen Zeitpunkt noch wenig ausgesagt werden, da Folgeprojekte mit veränderter Aufgabenstellung noch nicht abgeschlossen sind.

Danksagung

Wir danken Thomas Etzrodt, Gustav Pomberger, Dirk Schmelz, Andreas Spillner und den Gutachtern für nützliche Hinweise zu früheren Fassungen dieses Aufsatzes.

Literatur

1. Buhr, Raymond J.A.; Casselmann, Ronald S.: Architectures with Pictures. OOPSLA'92, SIGPLAN Not. 27,10 (1992) 466-483.
2. Ichbiah, J.D.; Barnes, J.G.P.; Heliard, J.C.; Krieg-Brueckner, B.; Roubine, O.; Wichmann, B.A.: Rationale for the Design of the ADA Programming Language. SIGPLAN Notices 14,6 (1979) Part B.
3. Møller-Pedersen, B., Haugen, O., Belsnes, D.: Introduction to Object Oriented SDL and to Elements of Corresponding Method, Working Paper of CCITT WP X/3, 1990
4. Borland C++ - Programmierhandbuch. Borland, Starnberg, 1992.
5. Borland Pascal mit Objekten 7.0 - Programmierhandbuch. Borland, Langen 1993.
6. Jacobson, Ivar; Christerson, Magnus; Jonsson, Patrik; Övergaard, Gunnar: Object-Oriented Software Engineering. Addison-Wesley, Wokingham usw., 1992.
7. de Champeaux, Dennis; Faure, Penelope: A comparative study of object-oriented analysis methods. JOOP 5,1 (1992) 21-33.
8. Dahl, Ole-Johan; Myhrhaug, Bjørn; Nygaard, Kristen: Common Base Language. Norwegian Computing Center, 1970.
9. Dießl, G.; Winkler, J.F.H.: Object-CHILL - An Object-Oriented Language for Systems Implementation. ACM Computer Science Conference '92, 139-147
10. Dumke, Reiner: Softwareentwicklung nach Maß. Vieweg, Braunschweig / Wiesbaden, 1992.
11. Ellis, Margaret A.; Stroustrup, Bjarne: The Annotated C++ Reference Manual. Addison Wesley, Reading 1990.
12. Johnson, Ralph E.; Foote, Brian: Designing Reusable Classes. JOOP 1,2 (1988) 22..30, 35.
13. Goldberg, Adele; Robson, David: Smalltalk-80. Addison Wesley, Reading etc., 1985.

14. Günther, W., Wackerbarth, G.: Designing ISDN Call Processing Software by Using Object-Oriented Techniques. International Switching Symposium 1992, Yokohama 25 - 30 October 1992, Vol.1, pp. 174..178.
15. Haake, G.: Der Testarbeitsplatz für die Systemsoftware des Kommunikationssystems Hicom. Informatik Spektrum 12, ? (1989) 331-336.
16. Lenzer, J,; Letschert, Th.; Lingen, A.; Hollis, D.: Eine Einführung in die Programmiersprache CHILL. Heidelberg, Hüthig 1987.
17. Hogrefe, Dieter: Estelle, LOTOS und SDL. Springer, Berlin 1989.
18. Introducing Ada 9X. Ada aktuell 1,2 (1993).
19. International Telecommunication Union (ed.): CCITT High Level Language (CHILL). Recommendation Z.200, Geneva 1988.
20. Jüttner, P.; Kolb, S.; Sieber, S.: Testen objektorientierter Software. In: Reichel, H. (Hrsg.): Informatik - Wirtschaft - Gesellschaft (23. Jahrestagung der GI). Springer, Heidelberg 1993, 427-432.
21. Kapsner, Franz: Software Reuse in Object-Oriented Programming Supported by a Class Library Management System. Proc. 10. IASTED, Acta Press, Zürich, 1992.
22. Kay, Alan C.: The Early History of Smalltalk. HOPL-II, SIGPLAN Not. 28,3 (1993) 69-95.
23. Meyer, Bertrand: Genericity versus Inheritance. OOPSLA'86, SIGPLAN Not. 21,11 (1986) 391-405.
24. Reiser, Martin; Wirth, Niklaus: Programming in Oberon. ACM Press, New York 1992.
25. Snyder, Alan: The Essence of Objects: Concepts and Terms. IEEE Software 10,1 (1993) 31..42.
26. Sammer, W.; Schwärtzel, H.: CHILL - Eine moderne Programmiersprache für die Systemtechnik. Berlin, Springer-Verlag 1982.
27. Weber, Franz: Towards a Discipline of Class Composition. OOPSLA'92 Addendum, OOPS Messenger 4,2 (1993) 149-151.
28. Wegner, Peter: Concepts and Paradigms of Object-Oriented Programming. OOPS Messenger 1,1 (1990) 7..87.
29. Wegner, Peter et al.: Object-Oriented Megaprogramming. OOPSLA'92, SIGPLAN Not. 27,10 (1992) 392-396.
30. Winkler, J.F.H.: Die Programmiersprache CHILL. Teil I, II. Automatisierungstechn. Praxis 28,5 (1986) 252..258 and 28,6 (1986) 290..294 .
31. Winkler, Jürgen F.H. : Erster Object-CHILL-Compiler fertig. Softwaretechnik-Trends 11,3 (1991) 219.
32. Winkler, Jürgen F.H. : "Class" considered harmful. CACM 35,8 (1992) 128..130.
33. Wirth, N.: Type Extensions. ACM TOPLAS, 10,2 (1988), 204-214.
34. CCITT Draft Recommendation Z.100: CCITT Specification and Description Language (SDL) and Annex A to the Recommendation, Geneva (1992).
35. CCITT Draft Recommendation Z.120: Message Sequence Chart (MSC); Geneva (1992).

05.02.19 16:06

Wolfgang Günther
Siemens AG ÖN ZL SW
Hoffmannstraße
81 359 München

Prof. Dr. Jürgen F.H. Winkler
Institut für Informatik
Friedrich-Schiller-Universität
07 740 Jena