Technische Hochschule Leipzig
Wissenschaftliche Zeitschrift
Jahrgang 15/1991 · Heft 1 · Seite 149–157

J. F. H. Winkler

Object-CHILL – Eine objektorientierte Erweiterung von CHILL

"The language designer ...
His task is consolidation, not innovation."
(C. A. R. Hoare [Hoa 73: 26])

Object-CHILL ist eine Erweiterung der Programmiersprache CHILL und enthält neben den Elementen der Objektorientierten Programmierung weitere innovative Sprachelemente, welche für die Erstellung großer Programme mit hohen Zuverlässigkeitsanforderungen wichtig sind. Dazu gehören: Unterstützung der Nebenläufigkeit durch nebenläufige Objekte, Unterstützung der Wiederverwendbarkeit durch generische Klassen und Unterstützung der Korrektheit durch Vor- und Nachbedingungen für Methoden und Invarianten für Objekte. Der erste Einsatz von Object-CHILL wird bei der Erstellung eines B-ISDN-Vermittlungssystems erfolgen.

Schlüsselwörter: CHILL, objektorientierte Programmierung, nebenläufiges Objekt, generische Klasse, Vor- und Nachbedingung, Invariante

1. Einführung

Die Programmiersprache CHILL [Rek 82; SS 82; Win 86; LLL 87] wurde beim CCITT (= Comité Consultatif International de Télégraphique et Téléphonique) entwickelt [ITU 8x] und auch international genormt [ISO/IEC 9496]. Ein Überblick über die Sprache wird in [Win 86] gegeben. CHILL ist zeitlich parallel zu Ada entstanden und hat auch einen ähnlichen Sprachumfang. Neben vielen Unterschieden im Detail besteht der Hauptunterschied darin, daß in Ada generische Programmeinheiten definiert werden können. Dies ist in CHILL nicht möglich.
Verwendet wird CHILL im wesentlichen im Bereich der Vermittlungstechnik [Pal 90; Sor 86]. Dort sind sehr große Systeme in CHILL entwickelt worden (Umfang mehrere Mio LOC), die weltweit eingesetzt werden. In der Bundesrepublik Deutschland sind derzeit alle im öffentlichen Telefonsystem eingesetzten rechnergestützten Vermittlungssysteme in CHILL programmiert. Da bereits etwa 300 Ämter mit solchen Anlagen ausgerüstet sind, werden sehr viele der täglichen Telefongespräche durch CHILL-Programme vermittelt.
Im Rahmen eines Versuchsprojektes wurde beschlossen, Vermittlungsprogramme in objektorientierter Technik zu entwickeln. Da CHILL bisher die Objektorientierung nicht unmittelbar unterstützt, wurde eine Spracherweiterung definiert [DSW 90], welche Objektorientierung, nebenläufige Objekte, generische Klassen und Vor- und Nachbedingungen für Methoden enthält und so die Erstellung großer Programme mit hohen Zuverlässigkeitsanforderungen [s. z. B. Neu 90, 90a] auf mehrfache Art und Weise unterstützt. In Object-CHILL lassen sich Einflüsse von Ada, C++ und Eiffel erkennen.
Der Aufsatz gibt einen kurzen Überblick über die wesentlichen, innovativen Aspekte von Object-CHILL. Die Kenntnis der prinzipiellen Aspekte der Objektorientierung wird dabei vorausgesetzt.

2. Klassen, Vererbung und Polymorphismus

2.1. Klassenspezifikation

Klassen werden in Object-CHILL als Verallgemeinerung von Moduln eingeführt, als Modultypen, analog zum Vorschlag in [Win 90] und zu den Task-Typen in Ada [Ref 83; Win 82]. Die Definition einer Klasse zur Definition der grundlegenden Eigenschaften eines Stack kann z. B. folgendermaßen aussehen:

```
IntStack: MODULE CLASS                                    /* Spezifikationsteil */
   GRANT  Empty, Full, Pop, Push  PREFIXED IntStack;        /* Exportklausel */
   Empty: PROC () RETURNS(Bool);  END Empty;          /* Methodenspezifikation */
   Full:  PROC () RETURNS(Bool);  END Full;
   Pop:   PROC (Elem Int LOC);  END Pop;
   Push:  PROC (Elem Int);  END Push;
   IntStack: CONSTR ();  END IntStack;                      /* Konstruktormethode */
/* Begin of private part ================== */       /* wird automatisch exportiert */
   SYN  StackLength = 127;                          /* Definition von Datenkomponenten */
   DCL  StackData  ARRAY(1:StackLength) Int;
   DCL TopOfStack  RANGE(0:StackLength);
END  IntStack;
```

Bild 1
Klassenspezifikation

In einer Klassenspezifikation werden vor allem die Methoden (= Prozeduren) definiert, die auf Objekte der betreffenden Klasse angewendet werden können. Außer Methoden kann eine Klasse auch Datentypen (in CHILL als „Modes" bezeichnet) und Konstanten exportieren.
Im obigen Beispiel werden die Methoden zur Manipulation von Stacks allgemein („Empty", „Full", „Pop" und „Push") exportiert. Außer den „normalen" Methoden gibt es noch zwei besondere Arten von Methoden: die Konstruktoren („CONSTR") und die Destruktoren („DESTR"). Konstruktoren werden automatisch ausgeführt bei der Erzeugung eines Objektes und dienen hauptsächlich der Initialisierung. Destruktoren werden automatisch ausgeführt bei der Vernichtung eines Objektes. Konstruktoren und Destruktoren haben denselben Namen wie die Klasse, zu welcher sie gehören.
Wie in Ada und C++ können Methoden überladen werden. Z. B. könnte man in obigem Beispiel einen zweiten Konstruktor definieren, welcher ein erstes Element in den Stack einspeichert:

```
IntStack: CONSTR (Elem Int);  END IntStack;
```

Die Datenkomponenten einer Klasse werden ebenfalls in der Klassenspezifikation definiert („StackLength", „StackData" und „TopOfStack"). Dies ist zwar aus logischen Gründen nicht erforderlich, vereinfacht aber die Realisierung der getrennten Übersetzbarkeit von Klassenspezifikation und Klassenrumpf (ähnlich wie der Private-Teil einer Paketspezifikation in Ada).

2.2. Klassenrumpf

Im Rumpf einer Klasse werden die Rümpfe der in der Klassenspezifikation definierten Methoden (ausgenommen die virtuellen Methoden) realisiert. Am Beispiel der oben spezifizierten Klasse „IntStack" kann das folgendermaßen aussehen.

```
IntStack: MODULE  BODY                                        /*  Klassenrumpf  */

  Empty: PROC () RETURNS (Bool);                          /*  Rumpf der Methode Empty  */
    RETURN (TopOfStack = 0);
  END  Empty;

  Full: PROC () RETURNS (Bool);
    RETURN (TopOfStack = StackLength);
  END  Full;

  Pop: PROC (Elem  Int LOC);
    IF NOT Empty()
    THEN Elem := StackData(TopOfStack);
         TopOfStack := TopOfStack -1;   FI;
  END  Pop;

  Push: PROC (Elem Int);
    IF NOT Full()
    THEN TopOfStack := TopOfStack +1;
         StackData(TopOfStack) := Elem;   FI;
  END  Push;

  IntStack:  CONSTR ();                                  /*  Rumpf des Konstruktors: */
    TopOfStack := 0;                                          /*  Initialisierung  */
  END  IntStack;

END  IntStack;
```

Bild 2
Klassenrumpf

2.3. Vererbung

Wie in der Objektorientierung üblich, können auch in Object-CHILL aus existierenden Klassen Unterklassen abgeleitet werden. Objekt-CHILL kennt zwei Formen der Vererbung: (1) IS-A-Vererbung (vollständige Vererbung mit Polymorphismus) und (2) INHERITS-Vererbung (selektive Vererbung ohne Polymorphismus).
Aus der allgemeinen Klasse „IntStack" kann z. B. eine Unterklasse „TraversableStack" [Maj 77] abgeleitet werden.
Bei der IS-A-Vererbung ist die Exportschnittstelle einer Oberklasse eine Teilmenge der Exportschnittstelle jeder ihrer Unterklassen. Daher erbt eine Unterklasse zuerst einmal die Exportschnittstelle ihrer Oberklasse. Diese Exportschnittstelle kann in der Unterklasse auf zweierlei Art modifiziert werden:
a) Hinzufügen weiterer Methoden („Down", „Up" im obigen Beispiel)
b) Redefinition ererbter Methoden (bei der Redefinition wird zu einer bereits vorhandenen Methode ein neuer Rumpf definiert; z. B. für die Methode „Push").

Die resultierende Exportschnittstelle von „TraversableStack" besteht nun aus:

```
Empty: PROC () RETURNS (Bool);  END Empty;
Full:  PROC () RETURNS (Bool);  END Full;
Down:  PROC ();  END Down;
Pop:   PROC (Elem Int LOC);  END Pop;
Push:  PROC (Elem Int);  END Push;
Up:    PROC ();  END Up;
TraversableStack: CONSTR();  END TraversableStack;
```

Tabelle 1

Class kind	Properties of Objects		
	Activity	Coordination of concurrent calls	Method Call
MODULE CLASS	passive	NO	SYNCHR
REGION CLASS	passive	YES	SYNCHR
CONCURRENT CLASS	active	YES	ASYNCHR

```
IntStack: REGION CLASS                                    /* Spezifikationsteil */
  GRANT  Empty, Full, Pop, Push  PREFIXED IntStack;        /* Exportklausel */
  /* weiter wie in Bild 1  */
END  IntStack;
```

```
Producer: CONCURRENT CLASS
     SEIZE CLASS IntStack;                                /* Importklausel */
     Producer: CONSTR(Receiver IntStack LOC);  END  Producer;
END Producer;

Producer: CONCURRENT BODY

     Producer: CONSTR(Receiver IntStack LOC);
        DCL  Message  Int;
        DCL Terminate  Bool;

        DO WHILE NOT Terminate;
            /*  . . .  */
            Receiver.Push(Message);              /* Synchroner Aufruf; wird bei */
            /*  . . .  */              /* Receiver mit anderen Aufrufen koordiniert */
        OD;
     END  Producer;
END  Producer;
```

```
Main: MODULE
    SEIZE CLASS IntStack, Producer;
    DCL Stack1  IntStack();
    DCL Producer1, Producer2  Producer(Stack1);          /* autom. Start von */
    ;                                                 /* Producer1 und Producer2 */
END  Main;
```

Bild 5
Benutzung von REGION- und CONCURRENT-Klassen

5. Generische Klassen

Neben den Möglichkeiten zur übersichtlichen Strukturierung von Programmen wird oft auch die Verbesserung der Wiederverwendbarkeit als einer der Vorteile der Objektorientierung genannt. Dies gilt aber nur für solche Arten der Wiederverwendung, welche durch das Vererbungsprinzip realisiert werden können. Andere Formen wie z.B. die allgemeine Parametrisie-

rung mit Typen, welches den generischen Programmeinheiten von Ada zugrunde liegt [End 88; Win 82], können durch das Vererbungsprinzip nicht realisiert werden [Mey 86]. Aus diesem Grunde enthält Object-CHILL auch generische Klassen, d.h. das Konzept der generischen Programmeinheiten ist in das Konzept der Objektorientierung integriert. Die technische Realisierung ist sehr ähnlich zu den generischen Programmeinheiten von Ada, lediglich die Anzahl der Arten von generischen Parametern ist etwas geringer (d.h. „constrained genericity" [Mey 86], während Eiffel eine Form von „unconstrained genericity" hat). Da auch Klassen als generische Parameter auftreten können, ergeben sich flexible Parametrisierungsmöglichkeiten.
Das folgende Beispiel zeigt die Definition einer generischen Klasse „StackTemplate", in welcher die Länge und der Elementtyp als generische Parameter auftreten.

```
StackTemplate: GENERIC                                    /* Generische Klasse */
   SYN StackLength = Int;                          /* Generischer Parameter: Konstante */
   SYNMODE ElemMode = ANY_ASSIGN;                    /* Generischer Parameter: Typ */
MODULE CLASS
   GRANT  Empty, Full, Pop, Push  PREFIXED StackTemplate;
   Empty: PROC () RETURNS(Bool);  END Empty;
   Full:  PROC () RETURNS(Bool);  END Full;
   Pop:   PROC (Elem ElemMode LOC);  END Pop;
   Push:  PROC (Elem ElemMode);  END Push;
   StackTemplate: CONSTR ();  END StackTemplate;
/* Begin of private part ============================== */
   DCL  StackData  ARRAY(1:StackLength) ElemMode;
   DCL  TopOfStack  RANGE(0:StackLength);
END   StackTemplate;
```

Bild 6
Generische Klasse

Aus einer generischen Klasse kann durch eine Ausprägung (generische Instantiierung) ein spezielles Exemplar, d.h. eine nichtgenerische Klasse, abgeleitet werden:

```
IntStack: MODULE CLASS = StackTemplate
   SYN StackLength = 127;                    /* aktueller generischer Parameter */
   SYNMODE ElemMode = Int;
END IntStack;
```

Durch diese Ausprägung entsteht die in Bild 1 dargestellte Klasse.
Will man für einen zweiten Elementtyp „Activation Record", der an anderer Stelle im Programm definiert sein möge, eine Stackklasse definieren, dann ist eine entsprechende Ausprägung zu formulieren:

```
ActRecStack: MODULE CLASS = StackTemplate
   SEIZE ActivationRecord;                                  /* Importklausel */
   SYN StackLength = 255;                    /* aktueller generischer Parameter */
   SYNMODE ElemMode = ActivationRecord;
END ActRecStack;
```

Das heißt, der Vorteil einer generischen Klasse zeigt sich vor allem bei mehrfacher Ausprägung.

6. Vor- und Nachbedingungen, Invarianten (Assertions)

Um die bisher recht große Lücke zwischen Spezifikation und Implementierung bei der Programmentwicklung zu verkleinern, wurde in Object-CHILL die Möglichkeit geschaffen, Me-

thoden mit Vor- und Nachbedingungen zu versehen und zu einer Klasse eine Invariante zu definieren, die dann jeweils für ein Objekt dieser Klasse gilt. Mit diesen Hilfsmitteln läßt sich die Semantik von Methoden zumindest teilweise formal spezifizieren. Vorbedingungen beschreiben in der Regel Bedingungen, denen die Eingabeparameter einer Methode genügen müssen, und die durch Typen allein nicht beschrieben werden können. Nachbedingungen beziehen sich entsprechend auf die Ausgabeparameter und ggfls. das Funktionsergebnis. Die Invariante beschreibt in der Regel die zulässigen internen Zustände eines Objektes.

```
Account :  MODULE  CLASS;                                    /*  Konto ohne Überziehung  */
    GRANT Balance, Deposit, Withdraw  PREFIXED Account;
    Account : CONSTR (InitialBalance  Int);
        PRE   InitialBalance > 0;                                  /*  Vorbedingung  */
        POST  Balance() = InitialBalance;  END Account;            /*  Nachbedingung  */
    Balance : PROC() RETURNS (Int);  END Balance;
    Deposit : PROC (Amount Int);
        PRE  Amount > 0;    END Deposit;
    Withdraw : PROC (Amount  Int);
        PRE  Balance() >= Amount AND Amount > 0;  END Withdraw;
/* Begin of private part ********************************** */
    DCL CurrBalance Int;
    INV  CurrBalance >= 0;                                         /*  Invariante  */
                                                  /*  Wird vor und nach Ausführung  */
END ACCOUNT;                                     /*  eines Methodenrumpfes überprüft  */
```

Bild 7
Assertions

Im vorliegenden Beispiel könnten zwar einige der Assertions durch eine andere Wahl der Datentypen ersetzt werden, wenn aber Account als Unterklasse eines allgemeineren Kontos definiert wird, welches auch Überziehen zuläßt, dann kann auf einige der Assertions nicht verzichtet werden.

LITERATUR

[Bri 75] *Brinch Hansen, P.*: The Programming Language Concurrent Pascal. IEEE Trans. on Software Engineering 1,2 (1975) 199...207.

[DSW 90] *Dießl, G.; Schulz, G.; Winkler, J. F. H.*: Object-CHILL: The Road to Object-Oriented Programming with CHILL. [Pal 90: 118...125].

[End 88] *Endres, A.*: Software-Wiederverwendung: Ziele, Wege und Erfahrungen. Informatik Spektrum 11,2 (1988) 85...95.

[Hoa 73] *Hoare, C. A. R.*: Hints on programming language design. Stanford Art. Intell. Lab., Memo AIM 224 October 1973 = CS-403.

[Hoa 74] *Hoare, C. A. R.*: Monitors: An Operating System Structuring Concept. CACM 17,10 (1974) 549...557.

[ITU 8x] International Telecommunication Union (ed.): CCITT High Level Language (CHILL). Recommendation Z.200, Geneva 1980, 1984, 1988.

[LLL 87] *Lenzer, J.; Letschert, T.; Lingen, A.*: Eine Einführung in die Programmiersprache CHILL. Huethig Verlag, Heidelberg, 1987.

[Maj 77] *Majster, M. E.*: Limits of the "Algebraic" Specification of Abstract Data Types. SIGPLAN Not. 12,10 (1977) 37...42.

[Mey 86] *Meyer, B.*: Genericity versus Inheritance. OOPSLA '86, SIGPLAN Notices, Vol. 21, No. 11 (1986) 391–405

[Neu 90] *Neumann, P. G.*: Risks to the Public in Computers and Related Systems. Software Eng. Notes 15,2 (1990) 3...22.

[Neu 90a] *Neumann, P. G.*: Inside Risks. CACM 33,7 (1990) 154.

[Pal 90] *Palma, A.* (Hrsg.): Proceedings of the 5th CHILL Conference, Rio de Janeiro, März 1990. (Wird bei Elsevier erscheinen)
[Ref 83] Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815A. Washington, 1983.
[Rek 82] *Rekdal, K.*: CHILL – The Standard Language for Programming SPC Systems. IEEE-COM 30,6 (1982) 1318...1328.
[Sor 86] *Sorgenfrei, H.* (Hrsg.): Proceedings of the 4th CHILL Conference, München, 1986.
[SS 82] *Sammer, W.; Schwärtzel, H.*: CHILL – Eine moderne Programmiersprache für die Systemtechnik. Springer, Berlin usw., 1982.
[Win 80] *Winkler, J.F.H.*: Das Prozeßkonzept in Betriebssystemen und Programmiersprachen I, II. Informatik Spektrum 2,4 (1979) 219...229 und 3,1 (1980) 31...40.
[Win 82] *Winkler, J.F.H.*: Ada: die neuen Konzepte. Elektron. Rechenanlagen 24,4 (1982) 175...186.
[Win 86] *Winkler, J. F. H.*: Die Programmiersprache CHILL. Automatisierungst. Praxis 28,5 (1986) 252...258 und 28,6 (1986) 290...294.
[Win 90] *Winkler, J. F. H.*: Adding Inheritance to Ada. In: Proc. of WADAS '90, 7th. Washington Ada Symposium, June 1990, 241...244.

Danksagung

Für wertvolle Hinweise zu einer früheren Fassung dieses Aufsatzes danke ich *M. Stadel*.

Verfasser:

Jürgen F. H. Winkler,
Siemens AG, Zentrale Forschung und Entwicklung,
München.

TECHNISCHE HOCHSCHULE LEIPZIG

WISSENSCHAFTLICHE ZEITSCHRIFT

**5. Symposium „Künstliche Intelligenz“
am 13. und 14. November 1990**

**Wissenschaftliche Symposium „Anwendung der Informatik
für Produkt- und Systemtechnologie“
am 13. und 14. Dezember 1990**

Jahrgang 15 · Heft 1/2 1991