**Peter Knauber**

# Wiederverwendbare Programmkomponenten mit Java

## 1. Einleitung

Für alle Arten von Programmiersprachen werden Übersetzer oder Interpreter benötigt, welche die Konstrukte der Sprache in Assembler oder direkt in Maschinensprache übersetzen, damit sie auf einer bestimmten Rechnerplattform ausführbar werden oder sie interpretativ abarbeiten. Aufgrund neuer Einsatzbereiche für Software und sich ändernder Vorgehensweisen beim Programmieren werden auch Modifikationen und Erweiterungen der Programmiersprachen nötig, was sich wiederum auf deren Übersetzer auswirkt.

Es fällt auf, daß viele Aufgaben, die ein Übersetzer leisten muß, immer wieder neu implementiert werden, obwohl genau diese Aufgaben für andere Sprachen bereits gelöst wurden. So haben z.B. viele imperative Sprachen einen großen gemeinsamen Kern von Sprachkonzepten. Dazu gehören Zuweisungen, bedingte und wiederholte Anweisungen sowie ähnliche Datentypen und Operatoren. Es ist sinnvoll, diese Aufgaben einmal zu realisieren und dann nur noch an die speziellen Gegebenheiten einer neu zu implementierenden Sprache anzupassen.

Im folgenden wird ein Verfahren vorgestellt, welches das Wiederverwenden bereits bekannter und implementierter Konzepte bei der Entwicklung von Übersetzern ermöglicht. Der Sprachentwickler paßt die Syntax der aktuellen Sprache an die Darstellungsform einer Bibliothek an, wobei er von Werkzeugen unterstützt wird. Dadurch wird auch das Eingliedern neuer problemspezifischer Konstrukte erleichtert.

Es liegt auf der Hand, daß ein entsprechendes System nicht schon vorab Realisierungen für alle möglichen Sprachkonstrukte enthalten kann. Selbstverständlich wird es immer möglich sein, neue Konstrukte zu definieren, für deren Übersetzung noch keine geeigneten Mechanismen zur Verfügung stehen. Daraus ergeben sich zwei Anforderungen an das Verfahren:

- Standardkonzepte, die zum Kern einer jeden Sprache gehören, sollen möglichst einfach und automatisiert an die aktuelle Sprache angepaßt werden können. Für „normale“ Sprachen kann dadurch der größte Teil des Sprachumfangs abgedeckt werden.
- Es muß möglich sein, Konstrukte für spezielle Aufgaben, die bisher noch nicht abgedeckt werden, einfach in einen Übersetzer einzufügen.

Um einen Ansatz zur Lösung dieser Anforderungen zu finden muß zunächst untersucht werden, inwiefern Standardverfahren für die Übersetzung von Programmiersprachen dafür geeignet sind.

## 2. Techniken zum Übersetzen von Programmiersprachen

### 2.1 Das traditionelle, phasenorientierte Übersetzermodell

Traditionell gliedert sich der Ablauf beim Übersetzen eines korrekten Programms in die Phasen lexikalische, syntaktische und semantische Analyse und Codegenerierung (vgl. [Wir86], [ASU88]). Ein (korrektes) Quellprogramm durchläuft während der Übersetzung nacheinander die verschiedenen Phasen, bevor als Ergebnis der Codegenerierung Zielcode erzeugt wird.

Dieses traditionelle Compilerdesign bringt jedoch einige wesentliche Nachteile mit sich und ist daher zum Lösen der Anforderungen aus Abschnitt 1 ungeeignet:

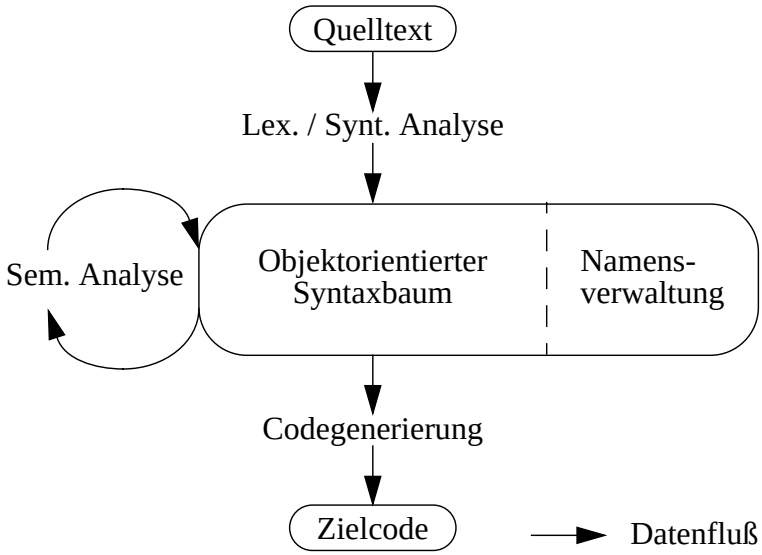
- Jedes Sprachkonstrukt muß in jeder Phase der Übersetzung behandelt werden. So liegen z.B. Informationen über die Repräsentation von Datentypen an vielen verschiedenen Stellen eines Übersetzers verteilt vor. Bei Modifikationen an der Sprache entstehen daher leicht Inkonsistenzen zwischen den verschiedenen Teilen eines Übersetzers.
- Bei der Integration neuer Sprachkonstrukte müssen viele verschiedene Programmteile bearbeitet werden. Dabei bleibt die Implementierung leicht unvollständig oder es treten ungewollte Seiteneffekte in anderen Programmteilen auf.
- Das Wiederverwenden von Compilerkomponenten ist nur sehr schwer möglich, da die Programmfragmente, die ein spezielles Konstrukt behandeln, nur sehr schwer lokalisiert und isoliert werden können.

### 2.2 Ein objektorientiertes Übersetzermodell

Von der objektorientierten Programmierung erhofft man sich besonders bezüglich Konsistenz und Wiederverwendbarkeit Vorteile gegenüber der klassischen imperativen Programmierung. Im Unterschied zur ablauforientierten imperativen Programmierung stehen bei einem objektorientierten Entwurf die Daten im Mittelpunkt. Das sind während der Übersetzung der abstrakte Programmbaum und die Namenstabellen bzw. deren Einträge. Ein objektorientierter Übersetzer arbeitet also wie folgt (vgl. Abbildung 1):

- Während der lexikalischen und der syntaktischen Analyse eines Programms wird der abstrakte Programmbaum in einer objektorientierten Repräsentation aufgebaut. Die entsprechenden Programmteile können automatisch aus der Syntax der Programmiersprache erzeugt werden.
- Die Routinen der semantischen Analyse und der Codegenerierung werden als Methoden realisiert und arbeiten auf den Objekten des abstrakten Syntaxbaums und den Einträgen der Namensverwaltung.

Abbildung 1: Aufbau eines objektorientierten Übersetzers




Dieser objektorientierte Ansatz löst die in Abschnitt 2.1 genannten Probleme:

- Die semantische Analyse und die Codegenerierung für ein Sprachkonstrukt wird durch Methoden realisiert, die in einer Klasse zusammengefaßt werden.
- Um neue Sprachkonstrukte zu integrieren muß nur eine neue Klasse mit den entsprechenden Methoden implementiert werden.
- Die Klassen sind die geeigneten Einheiten für die Wiederverwendung; um sie an eine spezielle Sprache anzupassen werden entsprechende Unterklassen (Spezialisierungen) gebildet. In Abschnitt 4 wird diese Technik durch ein Beispiel erläutert.

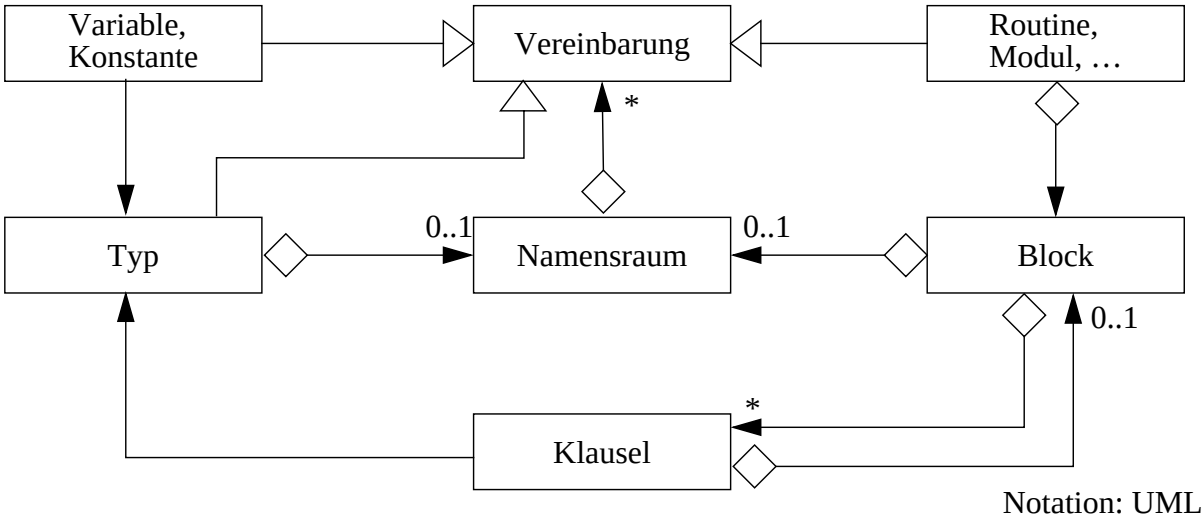
Es stellt sich nun die Frage, welche Sprachkonzepte sich für die Wiederverwendung eignen und wie sie organisiert werden sollen.

## 3. Konzepte von Programmiersprachen

In [Kna97] werden 18 verschiedene Programmiersprachen bezüglich ihrer gemeinsamen Konzepte untersucht. Es handelt sich um imperative und objektorientierte Vielzwecksprachen (*multi-purpose languages*) wie ALGOL, Pascal, C, C++ und Smalltalk-80, die entweder weit verbreitet sind bzw. waren oder aber durch ihre Konzepte andere Sprachen stark beeinflußt haben. Bei dieser Untersuchung zeigte es sich, daß die untersuchten Sprachen auf einige wesentliche Kernkonzepte reduziert werden können (Abbildung 2 veranschaulicht deren Zusammenhang):

- Namensräume stehen bei der Übersetzung eines Programms im Mittelpunkt. Sie verwalten eine beliebige Anzahl von Vereinbarungen, nach denen anhand eines Schlüssels (üblicherweise eines Namens) gesucht werden kann. Vereinbart werden können Variablen/Konstanten, Typen und Routinen/Module.
- Variablen und Konstanten sind spezielle Vereinbarungen. Beide können gleich behandelt werden, nur ist auf Konstanten kein schreibender Zugriff erlaubt. Beide werden mit einem bestimmten Datentyp vereinbart.
- Routinen und Module sind spezielle Vereinbarungen, sie werden hier stellvertretend für andere „Programm-Konstruktionen“ wie Prozeduren, Funktionen, Methoden und Programme aufgeführt.
- Der Rumpf von Routinen/Modulen wird durch einen Block gebildet. Diesem Block kann ein eigener Namensraum zugeordnet sein, der wiederum lokale Vereinbarungen enthalten kann.
- Ein Block enthält Anweisungen und Ausdrücke (Klauseln, engl. *Clause*), die wiederum eigene Blöcke beinhalten können. Sie liefern einen Wert und besitzen daher einen Typ (eventuell *void*).
- Auch Typen sind spezielle Vereinbarungen. Wie z. B. Records oder Klassen zeigen, können sie einen eigenen Namensraum besitzen.

Abbildung 2: Konzepte von Programmiersprachen




Diese Konzepte – sie werden im folgenden als *Kategorien*[1] bezeichnet – lassen sich nun in eine Bibliothek mit Compilerkomponenten umsetzen. Sie bilden damit die Basis für wiederverwend-

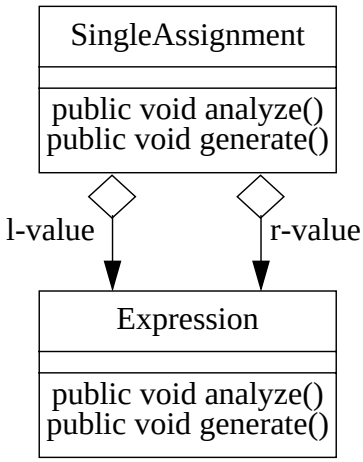
1. Der Begriff *Entwurfsmuster* (*design pattern*, s. [GHJV95]) wurde zugunsten von *Kategorien* bewußt vermieden. Die Elemente der einzelnen Kategorien sind in der Bibliothek als abstrakte Datentypen modelliert und daher untereinander austauschbar. Ihre Schnittstelle ist jedoch sehr problemspezifisch und damit für die Modellierung anderer Aufgabenstellungen nicht geeignet.

bare Komponenten von Übersetzern. Die Elemente der Kategorien wird man in einer objektorientierten Implementierung durch Klassen realisieren.

## 4. Ein Beispiel für die objektorientierte Übersetzung

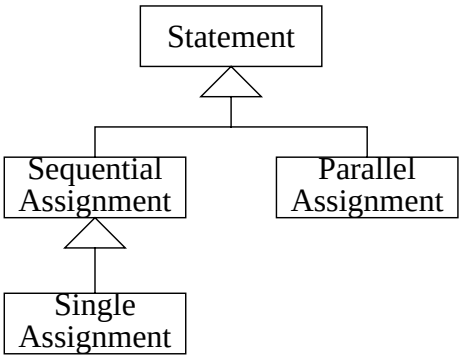
Abbildung 3 zeigt die Klasse *SingleAssignment* mit zwei Methoden *analyze* und *generate*, welche die semantische Analyse bzw. die Codegenerierung übernehmen. Eine Einfachzuweisung enthält zwei Ausdrücke (Klasse *Expression*): ein Zuweisungsziel (*l-value*) und eine Zuweisungsquelle (*r-value*). Wird nun die semantische Analyse der Zuweisung angestoßen, startet diese zunächst die Analyse der eingeschachtelten Ausdrücke, d.h. deren *analyze*-Methode. Sind diese semantisch korrekt, überprüft sie die Zuweisungskompatibilität und ob auf den *l-value* Schreibrecht, den *r-value* Leserecht besteht. Damit ist die semantische Analyse abgeschlossen, die Codegenerierung läuft analog ab. Es ist zu beobachten, daß für die Zuweisung nicht wichtig ist, ob es sich um einfache oder komplexe Ausdrücke handelt und wie diese aufgebaut sind. Benutzt werden ausschließlich Methoden der Schnittstellen *Clause*, *SemanticAnalysis* und *CodeGeneration* (vgl. Abschnitt 5 und Anhang A).

Abbildung 3: Übersetzung der Klasse *SingleAssignment*




Spezielle Konstrukte einer Programmiersprache können als Unterklassen, d.h. Spezialisierungen, existierender Klassen implementiert werden. Eine Einfachzuweisung wie in Abbildung 3 wird man z.B. als spezielle Form einer (sequentiellen oder parallelen) Mehrfachzuweisung realisieren (s. Abbildung 4). Statt zwei Liste von Ausdrücken als Zuweisungsquellen und -zielen tritt in einer Einfachzuweisung nur ein Paar von Ausdrücken auf, die restliche Semantik bleibt erhalten und wird von den Methoden der Oberklasse überprüft.

Abbildung 4: Vererbung bei Zuweisungen




Dieser Ansatz löst auch das Problem der Wiederverwendung von Compilerkomponenten für Sprachkonstrukte in anderen Übersetzern: Ganze Klassen können wiederverwendet und bei Bedarf durch Bilden von Unterklassen an die Bedürfnisse einer anderen Sprache angepaßt werden. Sie übernehmen dann für die (speziellen) Konstrukte der Sprache den größten Teil der semantischen Analyse und die Codegenerierung.

## 5. Das OCC-System und seine Realisierung in Java

Es liegt nun nahe, eine Bibliothek mit Klassen zu entwickeln, welche die Konzepte aus Abschnitt 3 widerspiegeln. Eine entsprechende Bibliothek wurde zusammen mit verschiedenen Werkzeugen zunächst als Prototyp in Common Lisp/CLOS [Ste90] entwickelt (s. [Kna97], [Tei96], [Ack96], [Hel97]). Dieses System wurde *OCC-System* (für Objectoriented Compiler Construction) genannt.

Es zeigte sich jedoch, daß der Prototyp recht langsam war und daß die verwendete Implementierungssprache auch andere Nachteile mit sich brachte. Daher wurden zunächst einige ausgewählte

Aspekte des Systems daraufhin untersucht, ob sie sich für eine Implementierung in Java eignen. Dabei fallen vor allem die folgenden Vorteile von Java auf:

- Für die Modellierung der verschiedenen Kategorien bieten sich die Interfaces von Java an. Sie legen eine Schnittstelle fest, die alle Objekte der entsprechenden Kategorie implementieren müssen (s. auch Anhang A). Das hat verschiedene Vorteile:
  - Der Compiler *javac* garantiert, daß nur dann Objekte einer Klasse gebildet und verwendet werden können, wenn diese die Schnittstelle ihrer Kategorie erfüllt.
  - Alle Klassen, die einer bestimmten Kategorie angehören, können über deren Schnittstelle adressiert werden. Insbesondere sind sie untereinander austauschbar. So ist es z.B. möglich, verschiedene Datentypen untereinander auszutauschen oder zusätzliche Datentypen zu einer Sprache hinzuzufügen, ohne den Rest des Compilers zu modifizieren.
  - Bei Erweiterungen der Bibliothek werden neue Sprachkonstrukte einer der Kategorien zugeordnet. Sobald sie deren Interface implementieren, können sie konsistent mit den bereits existierenden verwendet werden.

  Als Beispiel für eine Schnittstelle sind Auszüge aus dem Interface *Clause* in Anhang A angegeben.
- Die abstrakten Klassen von Java ermöglichen die Konstruktion einer Hierarchie von Klassen, die exakt die Beziehungen zwischen den Konstrukten der untersuchten Programmiersprachen widerspiegeln.
- ABstrakte Methoden ermöglichen es, die Klassen der Bibliothek an die konkrete Syntax einer beliebigen Programmiersprache anzupassen.
- Werden neue Kategorien gebildet (z.B. für andere Arten von Programmiersprachen), so können diese die Klassen bereits existierender Kategorien verwenden.
- Das Werkzeug *javadoc* eignet sich hervorragend für die Dokumentation einer Klassenbibliothek, da es genau dafür entwickelt wurde. Es ersetzte ein entsprechendes Werkzeug, das speziell für die Common Lisp-Implementierung entwickelt worden war.

  Die erzeugte Dokumentation erleichtert dem Benutzer der Bibliothek die Orientierung innerhalb des Klassensystems und die Lokalisierung geeigneter Kandidaten für die Wiederverwendung.

## 6. Zusammenfassung

Der Ansatz des objektorientierten Übersetzerbaus erlaubt (zumindest im Bereich imperativer und objektorientierter Sprachen) einen hohen Grad von Wiederverwendung bereits existierender Compilerkomponenten und ermöglicht die einfache Integration neuer Konzepte. Es zeigte sich z.B., daß ein Übersetzer für die Sprache Oberon-2 [MW92] in nur sechs Wochen anstelle von typischerweise sechs Monaten bei konventionellen Implementierungen (s. [AZM94], [Lut94]) zur Verfügung stand. Dabei wurden vier Wochen für Oberon-2-spezifische Konzepte benötigt, der gesamte Rest der Sprache konnte in zwei Wochen implementiert werden.

Die Sprache Java bietet geeignete Konzepte, um den Ansatz in ein lauffähiges System umzusetzen. Die Sprache erweist sich dabei als flexibel genug, um einen generischen und damit erweiterbaren Ansatz zu realisieren. Andererseits bietet Java geeignete Kontrollmöglichkeiten (z.B. ein statisches Typsystem), um den korrekten Einsatz vorhandener Programmkomponenten zu gewährleisten. Die Entwicklungsumgebung der Sprache enthält Werkzeuge, welche die Entwicklung, Wartung und Dokumentation großer Klassensysteme erleichtern.

**Anhang A: Auszüge aus der Schnittstelle der Kategorie *Clause***

Als Beispiel dient hier die Schnittstelle für die Kategorie *Clause*, die durch das Interface *Clause* beschrieben wird. Die Schnittstellen *SemanticAnalysis* und *CodeGeneration* fordern Routinen für die semantische Analyse und die Codegenerierung, sie müssen von allen Klassen der Bibliothek implementiert werden.

```
/** Dieses Interface beschreibt die Schnittstelle von Objekten
  * der Kategorie "Clause". … */
public interface Clause extends SemanticAnalysis, CodeGeneration
{
   /** Liefert den Typ eines Wertes.
     * Diese Methode liefert den Typ eines Wertes als Ergebnis.
     * Dabei handelt es sich immer um ein Objekt der Klasse
     * <A HREF="TypeConstructor.html">TypeConstruktor</A>. */
   public      Type        typeOf ();

   /** … */
   public      void        checkAccess ( AccessRight kind );

   /** … */
   public      boolean     isConstant ();

   /** … */
   public      Position    getPosition ();

} // Clause
```

Alle Werte (s. Abschnitt 3) erfüllen dieses Interface und werden ausschließlich durch dessen Methoden adressiert. Lediglich spezielle Konstrukte einer Sprache können diese Schnittstelle erweitern um sprachspezifische Informationen auszutauschen. Die entsprechenden Klassen sollten dann jedoch als *final* vereinbart werden.

**Literatur**


Ack96 M. van Acken: *Entwurf und Implementierung eines Klassensystems für die semantische Analyse und Codegenerierung von Namensräumen*, Diplomarbeit Universität Kaiserslautern, Fachbereich Informatik, 1996

ASU88 A. V. Aho, R. Sethi, J. D. Ullman: *Compilerbau*, Addison-Wesley, Bonn, 1988

AZM94 M. van Acken, J. Maurer, J. Zimmermann: *Entwurf und Implementierung einer Programmierumgebung für die Erstellung von Oberon-2-Programmen und ihre Übersetzung nach C++*, Projektarbeit Universität Kaiserslautern, Fachbereich Informatik, 1994

GHJV95 E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995

Hel97 N. Helbrich: *Entwurf und Implementierung eines Klassensystems für die semantische Analyse und Codegenerierung von Anweisungen*, Diplomarbeit Universität Kaiserslautern, Fachbereich Informatik, 1997

Kna97 P. Knauber: *Ein System für die Konstruktion objektorientierter Übersetzer*, Dissertation Universität Kaiserslautern, Fachbereich Informatik, 1997

Lut94 A. Lutz, *Entwurf und Implementierung eines objektorientierten Compilers am Beispiel der Übersetzung von Oberon-2 nach C++*, Diplomarbeit Universität Kaiserslautern, Fachbereich Informatik, 1994

MW92 H. Mössenböck, N. Wirth: *The Programming Language Oberon-2*, Institut für Computersysteme, ETH Zürich, 1992

Ste90 G. L. Steele Jr.: *Common Lisp: The Language, 2nd Edition*, Digital Press, 1990

Tei96 M. Teichmann: *Entwurf und Implementierung eines Klassensystems für die semantische Analyse und Codegenerierung von Typen*, Diplomarbeit Universität Kaiserslautern, Fachbereich Informatik, 1996

Wir86 N. Wirth: *Compilerbau*, 4. Auflage, Teubner Studienbücher Informatik, Stuttgart, 1986


**Autor:**


Dr. rer. nat. Peter Knauber
Universität Kaiserslautern, Postfach 3049
67653 Kaiserslautern
Tel.: 0631 - 205-2642, Fax: 0631 - 205-2803
e-mail: knauber@informatik.uni-kl.de