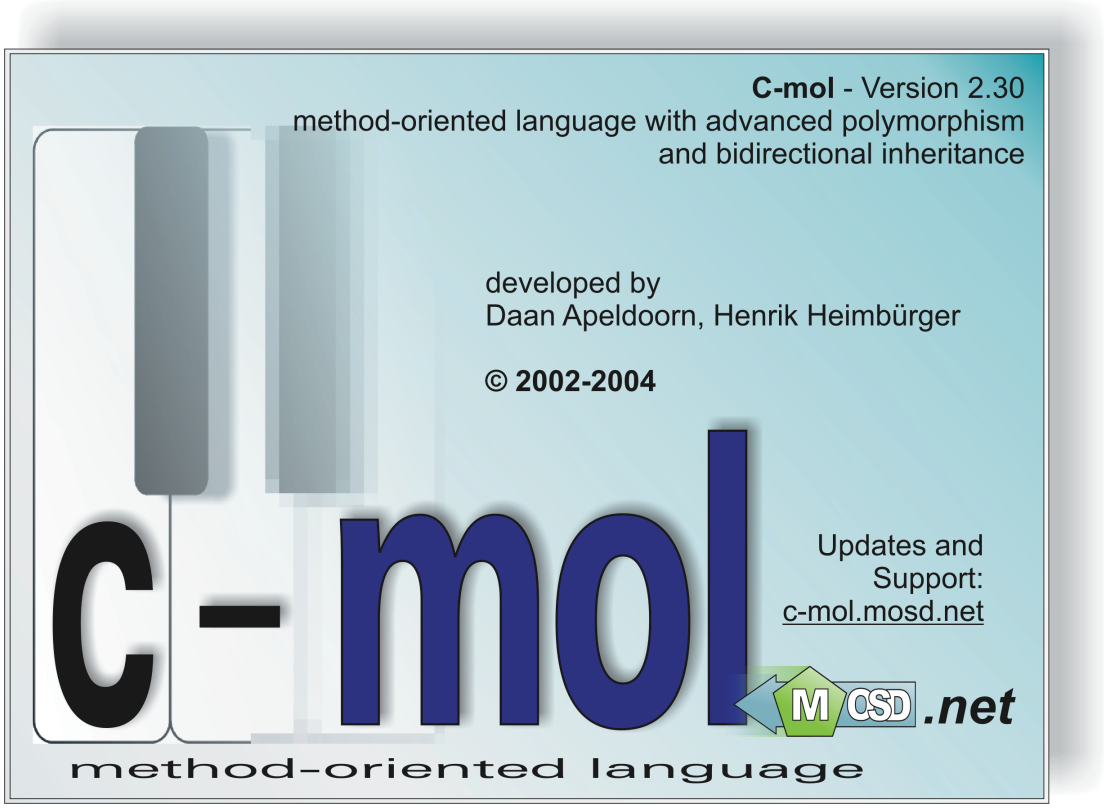
Daan Apeldoorn • Henrik Heimbürger



Methodenorientierte Softwareentwicklung mit der Programmiersprache C-mol

Ein neues Konzept für effizientere Entwicklung und Implementierung von Softwaresystemen

v3.1, September 2004

Inhaltsverzeichnis

Einleitung 5

1. Methodenorientierte Softwareentwicklung

1.1 Grundlegende Ideen und Ansätze 7

1.2 Konzeptionelle Umsetzung 9

1.3 Bewertung 14

2. Die Sprache C-mol

2.1 Einführung in C-mol 20

2.2 Sprachkonzepte und Syntax 21

3. Der C-mol-Compiler

3.1 Compiler versus Präprozessor 31

3.2 Architektur des Compilers 32

3.3 Scanner 34

3.4 Parser 35

3.5 Das Fehlerrückführungssystem 39

4. C-mol in der Praxis

4.1 Integration in Entwicklungsumgebungen 42

Anhang A: Beispielprogramm 46

Anhang B: C-mol-Grammatik in EBNF 50

Anhang C: Abbildungsverzeichnis / Quellenangaben 52

Einleitung

In den letzten Jahrzehnten wurden Softwareprojekte zunehmend umfangreicher und komplexer. Dies führte in der Vergangenheit zur Entwicklung neuartiger Entwicklungsansätze und -strategien, um den wachsenden Umfang und die steigende Komplexität von Software auch weiterhin entwicklungstechnisch erfassen und verwalten zu können.
Eine der größten Neuerungen während dieser Zeit ist die Idee der objektorientierten Softwareentwicklung, welche die bis dato meist strukturierte Entwicklung u.a. hinsichtlich Modularisierung und Code-Wiederverwendbarkeit revolutionierte. So stellt die Objektorientierung heute ein konsistentes und sauberes System zum Entwurf und – mit ihren Sprachvertretern wie z.B. C++ und Java – auch zur Implementierung von Softwareprojekten dar.
Doch auch seit der Entwicklung objektorientierter Programmierung ist das Wachstum von Softwareprojekten nicht zum Stillstand gekommen. Dies wirft erneut die Frage nach der Existenz und der Notwendigkeit neuer Wege auf, um die moderne Softwareentwicklung noch effizienter und strukturierter gestalten zu können. Diese neuen Wege sollten gemäß einer evolutionären Wissenschaft und unter Berücksichtigung wirtschaftlicher Situationen beschritten werden können, ohne Altbewährtes dabei verwerfen zu müssen.

Mit dieser Arbeit möchten wir das von uns entwickelte Konzept der methodenorientierten Softwareentwicklung (MOSD; method-oriented software development) und dessen Funktionsweise bis hin zur praxistauglichen Umsetzung in der von uns entwickelten methodenorientierten Programmiersprache ‚C-mol' darlegen und damit der Fachwelt näher bringen.
Die methodenorientierte Programmierung ist dabei als ein konsistentes und sauberes System anzusehen, welches die Effizienz der Softwareentwicklung vom Entwurf bis zur Implementierung insbesondere hinsichtlich der Vermeidung von redundanten Implementierungen verbessern soll, ohne jedoch die altbewährten objektorientierten Konzepte außer Acht zu lassen. Vielmehr soll in dieser Arbeit dargelegt werden, dass höchste Effizienz durch eine Hybridisierung beider Ansätze erreicht werden kann. Alle methodenorientierten Konzepte sind dabei derart äquivalent zu denen der objektorientierten Entwicklung aufgebaut, dass der Einstieg für objektorientierte Entwickler und Programmierer „fließend" und ohne größeren Lernaufwand in kürzester Zeit erfolgen kann.

Mit der Sprache (bzw. der Spracherweiterung) C-mol und dem als proof-of-concept dienenden, jedoch voll einsatzfähigen C-mol-Compiler ist die methodenorientierte Softwareentwicklung bereits heute in der Praxis einsetzbar. Auf Grund der flexiblen Struktur des Compilers ist es möglich C-mol auf verschiedene C++-Compiler aufzusetzen und in verschiedene Entwicklungsumgebungen zu integrieren. So kann auf den meisten Plattformen Software methodenorientiert entwickelt und in C-mol implementiert werden.

1. Methodenorientierte Softwareentwicklung

1.1 Grundlegende Ideen und Ansätze

Die methodenorientierte Softwareentwicklung stellt ein neues und praxisnahes Konzept zur Verbesserung moderner Softwareentwicklungsprozesse dar. Dabei sollen jedoch die objektorientierten Ideen nicht in den Hintergrund gestellt werden, sondern vielmehr höchste Effizienz durch Kollaboration mit den methodenoorientierten Konzepten erzielen.
Der Schwerpunkt liegt vor allem bei der Reduzierung von Code durch implizite Vermeidung redundanter Implementierungen und dem Zugewinn an Übersichtlichkeit und die damit verbundene Vereinfachung der Codewartung durch verbesserte Modularisierung. Die methodenorientierte Softwareentwicklung und Programmierung (auch MOP genannt) bietet dafür eine Reihe neuer Ideen und Ansätze für Entwicklung und Implementierung.

Während mit der objektorientierten Entwicklung und Programmierung (auch OOP genannt) versucht wird, ein softwaretechnisch zu lösendes Problem in „Objekte“ zu gliedern, die in Form von Klassen und ihrer Instanzen (Objekte) im Quelltext abgebildet werden, ist die grundlegende methodenorientierte Idee, ein solches Problem in seine „Tätigkeiten“ zu zerlegen, welche im Code durch sogenannte Methoden repräsentiert werden. In Methoden werden bestimmte Behandlungsroutinen (sogenannte Handlings) definiert, welche festlegen, wie mit den Objekten bestimmter Typen beim Aufruf einer Methode verfahren werden soll. Wie auch beim objektorientierten Ansatz existieren selbstverständlich auch in der MOP entsprechende Konzepte zur Code-Wiederverwendung (durch Vererbung („Aufleiten“) von Methoden) und Polymorphie (durch polymorphe Methoden). All diese Konzepte werden in den folgenden Kapiteln ausführlich erläutert.

Durch die primäre Orientierung der Planung und des Entwurfs an den Methoden entstehen Vorteile wie die Vermeidung redundanter Implementierung und eine stärkere Modularisierung *implizit*, d.h. ohne dass dies gezielt berücksichtigt werden muss. So entsteht eine vollständige Trennung zwischen Entwurf und Implementierung der Software, bei der nun keine zyklischen Beziehungen zwischen Entwurfs- und Implementierungsschichten mehr notwendig sind. Des Weiteren muss durch die Vermeidung von Coderedundanzen weniger Code erstellt werden und Codewartungen können gezielter erfolgen, was eine Erhöhung der Produktivität des Softwareentwicklungsprozesses zur Folge hat.

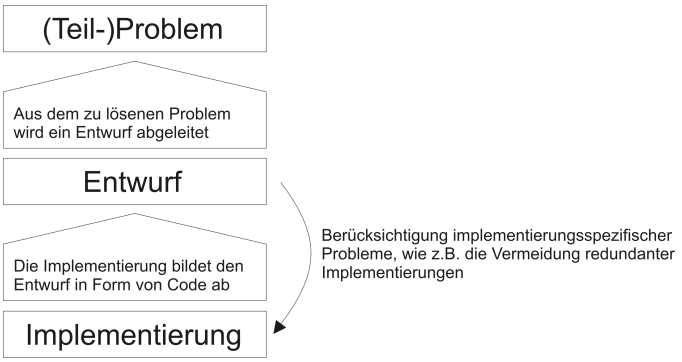


Abb. 1: Typischer Entwicklungsprozess vom realen Problem zur Implementierung

Erstrebenswert wäre hier ein Entwurfsverfahren, welches automatisch implementierungsspezifische Probleme, wie die Vermeidung redundanter Implementierungen umgeht, ohne dass dies explizit beim Entwurf berücksichtigt werden muss. Das Entwurfsverfahren sollte idealerweise schon eine möglichst optimale Implementierung mit sich bringen.

So wird beim Verfolgen des methodenorientierten Paradigmas ein Entwicklungsprozess möglich, der geradlinig vom realen Problem zu dessen Abbildung in Form von Code führt. Vor allem im Rahmen der Hybridisierung mit dem objektorientierten Paradigma wird in einer modularen Art und Weise sowohl eine saubere Komposition (bottom-up) als auch eine Dekomposition (top-down) von Problemen ermöglicht, so dass eine adäquate Strategie in Abhängigkeit des zu lösenden softwaretechnischen (Teil-)Problems gewählt werden kann.

1.2 Konzeptionelle Umsetzung

So wie in der OOP die Klasse, so ist in der MOP die Methode der Ausgangspunkt jeder Entwicklung. Methoden sollen mit eingehenden Daten umgehen. Dafür werden sogenannte Handlings in jeder Methode definiert. Methoden bündeln somit verwandte Codeteile an einer Stelle im Quelltext und stellen modulare Einheiten mit einer klar definierten Schnittstelle dar.
Jedes Handling steht für einen (oder mehrere) Datentyp(en) und bestimmt, wie beim Aufruf der Methode mit einem Datum dieses Datentyps (bzw. dieser Datentypen) umgegangen werden soll. Methoden sind somit ad-hoc polymorph für alle Datentypen, deren Behandlungen durch die Handlings der Methode definiert sind.
Im Gegensatz zu Klassen müssen keine Instanziierungen von Methoden vorgenommen werden, da es sich um reine „Codeobjekte“ und nicht um Daten handelt.

1.2.1 Handlings, Prolog und Epilog

Jedes Handling einer Methode besitzt eine sogenannte Handling-Identifikation, die aus einem (oder mehreren) Datentypen besteht und festlegt, welche Typen das Handling behandeln kann. So kann ein Handling also auch zur Behandlung mehrerer Typen definiert werden, sofern diese eine identische Behandlung in der Methode erfahren sollen (sogenannte Multi-Handlings). Dies dient der Vermeidung von mehrfach implementierten Routinen für verschiedene Typen und kann so helfen, Klassenhierarchien zu vereinfachen.
Jedes Handling besitzt eine individuelle Parameterliste, so wie dies auch von Methoden aus der OOP bekannt ist. Ferner besitzt jede Methode eine Methodenparameterliste, welche an jedes in der Methode enthaltene Handling übergeben wird. Die Methodenparameterliste stellt somit eine „Meta-Schnittstelle“ für die gemeinsamen Parameter aller in der Methode enthaltenen Handlings dar – ein weiteres Element der Zentralisierung.

Jede Methode kann zwei spezielle „Handlings“ definieren: einen sogenannten Prolog und einen sogenannten Epilog. Diese werden stets vor, respektive nach dem Ausführen eines jeden Handlings der Methode aufgerufen und können somit auch als gemeinsame Handlings für alle Datentypen, welche die Methode verarbeiten kann, betrachtet werden. Auch Prolog und Epilog einer Methode erhalten deren Methodenparameter. Sie stellen die Äquivalente zum Konstruktor und Destruktor von Klassen in der OOP dar.
Codeteile, welche für alle Handlings einer Methode identisch sind, können so also in Prolog und Epilog zusammengefasst werden.

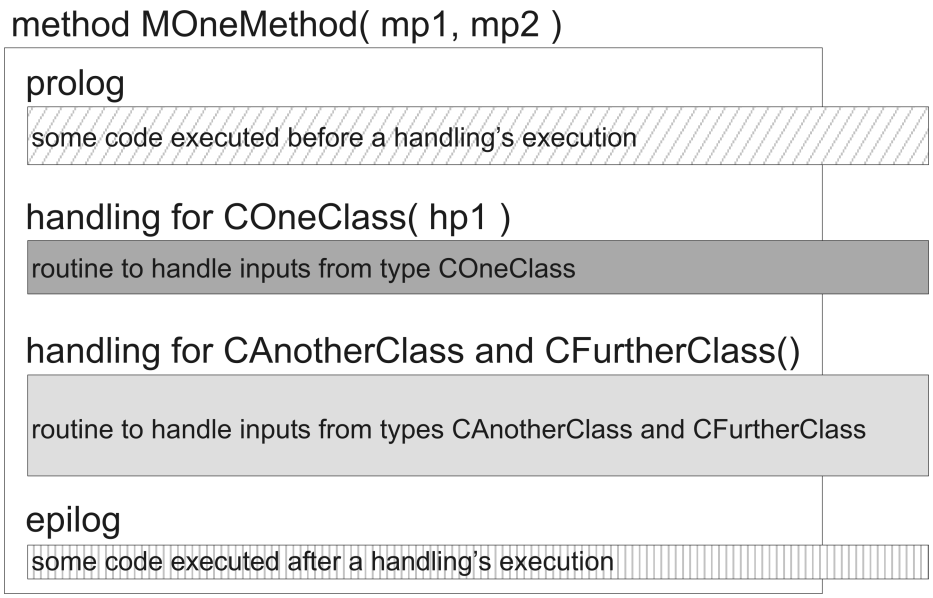


Abb. 2: Eine einfache Methode

Die Methode ‚MOneMethod' besitzt neben Prolog und Epilog ein Handling mit dem Parameter ‚hp1' zur Behandlung von Objekten des Typs ‚COneClass' sowie ein Multi-Handling mit leerer Parameterliste zur Behandlung von Objekten der Typen ‚CAnotherClass' und ‚CFurtherClass'.
Die Methodenparameter ‚mp1' und ‚mp2' werden an jedes Handling, sowie an Prolog und Epilog weitergereicht.

In jedem Handling, sowie in Prolog und Epilog kann auf die zu behandelnden Daten zugegriffen werden. Dieser sogenannte Methoden-Input ist in Multi-Handlings sowie in Prolog und Epilog ad-hoc polymorph und nimmt einen sogenannten Schnittmengen-Datentyp an.
Er besitzt im Falle von Multi-Handlings die Menge aller Eigenschaften welche durch die Verknüpfung aller in der Handling-Identifikation aufgeführten Typen mit einem logischen ‚und' gebildet wird.
Im Falle von Prolog und Epilog besitzt der Methoden-Input die Menge aller Eigenschaften, welche durch die Verknüpfung aller von der Methode zu behandelnden Datentypen mit einem logischen ‚und' entsteht.

1.2.2 Erweiterter Polymorphismus

Wie bereits aus dem vergangenen Kapitel hervorgeht, sind Methoden ad-hoc polymorph für alle Datentypen, deren Behandlungen durch die Handlings der Methode definiert sind. Auch ein einzelnes (Multi-)Handling einer Methode kann ad-hoc polymorph sein, indem es für mehrere Datentypen – die eine identische Behandlung erfahren sollen – definiert wird. Jedoch wird für moderne und flexible Programmstrukturen auch ein polymorpher Umgang für zur Entwicklungszeit noch nicht exakt bekannte Typen gefordert. Dieses in objektorientierten Ansätzen meist über das Ableiten von Klassen und die Implementierung aus (abstrakten) Basisklassen stammenden virtuellen oder rein-virtuellen (OOP-)Instanzmethoden realisierte Konzept, ist in der MOP über die Methoden durch die Definition virtueller oder rein-virtueller Handlings umgesetzt.

Äquivalent zu Klassen wird eine Methode polymorph, sobald sie eines oder mehrere virtuelle Handlings besitzt. Eine solche Methode kann dann als „noch zur Laufzeit existent“ betrachtet werden, was bedeutet, dass sie auch für zur Entwicklungszeit noch nicht exakt bekannte Typen die Selektion eines korrekten Handlings und damit die korrekte Verarbeitung von deren Instanzen gewährleisten kann.
Somit ist es auch möglich Polymorphie auf Klassen verschiedener, unvereinter Klassenhierarchien auszuüben, um beispielsweise verschiedene im Projekt eingesetzte Bibliotheken einfach zusammenführen, anpassen und erweitern zu können.
Ein weiterer Vorteil aus dieser der OOP gegenüber erweiterten Form der Vielgestaltigkeit ist die uneingeschränkte Verfügbarkeit aller in den *spezialisierten* Klassen vorhandenen Attributen und Methoden an jeder Stelle (auch wenn dort der exakte Typ zur Entwicklungszeit noch nicht bekannt ist), da in den Handlings für die spezialisierten Klassen natürlich stets der konkret ausgeprägte Typ bekannt ist.

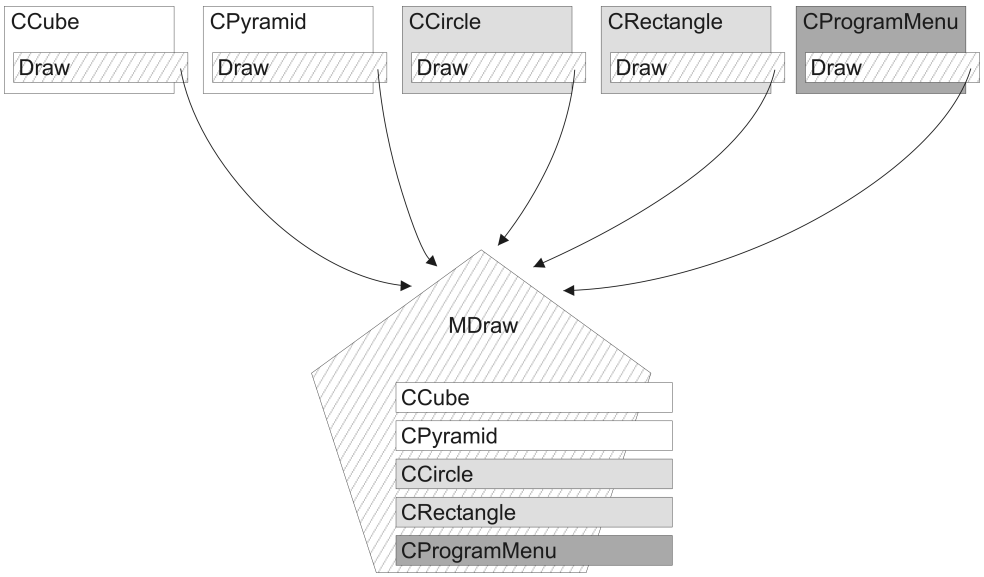


Abb. 3: Beispiel für erweiterten Polymorphismus

Klassen mit gemeinsamen Grauschattierungen haben hier identische Basisklassen. Die Methode ‚MDraw‘ kann jedes dieser Objekte verarbeiten, unabhängig davon, ob sie einer gemeinsamen Basisklasse angehören und ob ihr exakter Typ zum Zeitpunkt des Aufrufs der Methode bekannt ist oder nicht.

1.2.3 Bidirektionale Vererbung

Als Äquivalent zur Ableitung von Klassen in der OOP sieht die MOP das Konzept der Vererbung in Form der Aufleitung von Methoden vor. Während das Ableitungskonzept der OOP ermöglicht, Klassen zu spezialisieren und zu erweitern, dient die Aufleitung von Methoden neben der Erweiterung vor allem der Generalisierung einer oder (im Falle der Mehrfachvererbung von Methoden) mehrerer Methoden[1]. So ist es möglich im Rahmen der Kollaboration mit dem objektorientierten Vererbungsmechanismus stets die adäquate Richtung zur Komposition bzw. Dekomposition eines (Teil-)Systems einzuschlagen.

Der Vererbungsmechanismus spielt in der MOP eine Schlüsselrolle, da er das durch „Methoden als modulare Einheiten“ vorgegebene Modell zu einem in sich konsistenten System schließt:
Eine Integralmethode erbt alle Handlings einer Integrationsmethode, wobei auch die Methodenparameter mit vererbt werden. Prolog und Epilog der Integrationsmethode werden für jedes Handling weiterhin implizit aufgerufen, auch wenn dieses über die Integralmethode aufgerufen wird.

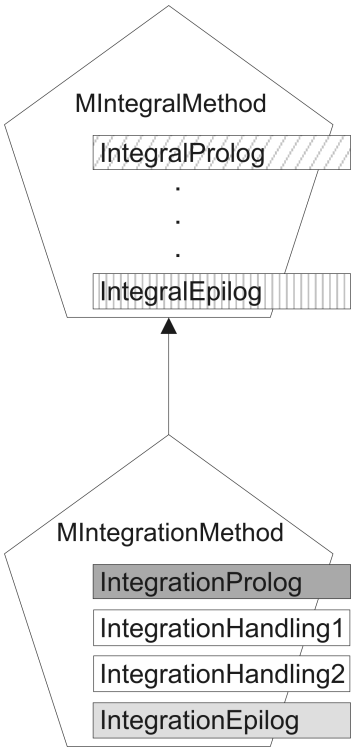


Abb. 4a: Methodenvererbung (UML-ähnliche Darstellung)

Aufleiten einer Integrationsmethode zu einer Integralmethode, dargestellt im UML-ähnlichen Stile.

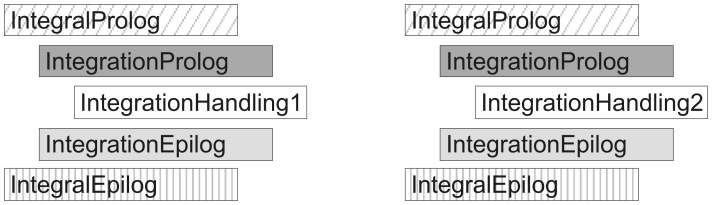


Abb. 4b: Staffelung der Prologe und Epiloge bei der Vererbung von Methoden

Darstellung der Aufrufreihenfolge der Prologe und Epiloge von Integrations- und Integralmethode bei einer Vererbung wie in Abb. 3a dargestellt.

[1] Die zu generalisierenden Methoden werden in Anlehnung an die mathematische Terminologie als Integrationsmethoden bezeichnet; die Obermethoden, zu welchen die Integrationsmethoden generalisiert werden, als Integralmethoden. Analog dazu sprechen wir bei vererbten Handlings auch gelegentlich von Integrationshandlings.

Der Prolog und der Epilog der Integralmethode stellen dabei einen „Meta-Prolog“ und „Meta-Epilog“ für alle Integrationsmethoden dar und dienen somit der Vermeidung redundanter Implementierungen in den Prologen und Epilogen sowie den Handlings aller Integrationsmethoden. Somit kann mit Hilfe des Vererbungsmechanismus das Konzept der Vermeidung redundanter Implementierungen durch Einsatz von Prolog und Epilog auf höheren Abstraktionsebenen beliebig fortgesetzt werden.
Wie auch (OOP-)Methoden bei der Ableitung von Klassen, so können auch einzelne Handlings bei der Aufleitung von Methoden überschrieben werden.
Als Gedankenstütze ist es auch möglich, eine Integrationsmethode mit all ihren Handlings als „ein“ Handling in der Integralmethode zu betrachten.

Es existiert ebenso ein Konzept zur Definition abstrakter Schnittstellen, welches in der OOP meist durch abstrakte Klassen und abstrakte oder rein-virtuelle Methoden, die es im Zuge der Ableitung dieser Klassen zu implementieren gilt, realisiert wird. So ist es äquivalent zur OOP auch in der MOP möglich abstrakte Methoden[1] als Schnittstellen zu definieren, welche abstrakte oder rein-virtuelle Handlings enthalten, die im Zuge der Aufleitung dieser Methoden implementiert werden müssen.

1.2.4 Methodenaufrufe und -verkettung

Ein Methodenaufruf besteht aus einem Methodennamen, einer Methodenargumentliste, einem Methodeninput sowie einer Handlingargumentliste. Der Methodenname identifiziert dabei die aufzurufende Methode, die Methodenargumentliste enthält die an die Methode zu übergebenden Argumente.
Der Methodeninput legt die Selektion des entsprechenden Handlings fest, die Handlingargumentliste enthält die an das selektierte Handling zu übergebenden Argumente.

Weiterhin ist es möglich Methoden zu verketten. Solche sogenannten Methodenketten werden sequentiell ausgeführt, indem ein Methodeninput die Methodenkette passiert. Dies ist stark an das Konzept des „Pipings“ angelehnt und dient der Zusammenfassung zusammengehöriger Operationen in einer einfachen und übersichtlichen Art und Weise.

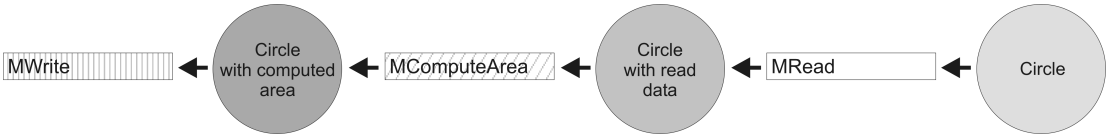


Abb. 5: Verkettung von Methoden

Das Input-Objekt passiert hier die drei Methoden sequentiell, wobei es sich nach dem Passieren jeder einzelnen Methode in einem anderen Zustand befindet (ausgenommen die Write-Methode, sie verändert den Zustand des Objekts nicht).

[1] Im Folgenden der Eindeutigkeit halber auch gelegentlich Schnittstellen- oder Interface-Methoden genannt.

1.3 Bewertung

In diesem Kapitel sollen nun die Qualitäten der in den vergangenen Kapiteln theoretisch dargelegten Ansätzen der MOP erläutert werden.
Dies soll insbesondere unter dem Aspekt der in der Praxis entstehenden Effizienzsteigerung des Softwareentwicklungsprozesses geschehen. Die praktische Umsetzung der Konzepte mit Hilfe der Sprache C-mol wird dann Gegenstand der folgenden Kapitel sein.

Bei allen im Folgenden genannten Vorteilen gegenüber konventionellen Entwicklungsstrategien liegt der Schwerpunkt darauf, dass diese implizit, d.h. ohne explizite Berücksichtigung bei den Planungs- und Entwurfszyklen des Projektes entstehen. Vielmehr sind sie durch die primäre Orientierung der Entwicklung an den Methoden eines Systems automatisch vorgegeben.
Dies sollte auch aus wirtschaftlicher Sicht von Interesse sein, da gerade in der Wirtschaft häufig weder ausreichend Zeit noch Geld in einen sauberen und weitsichtigen Softwareentwurf investiert wird, am Ende jedoch alle Vorzüge eines solchen – wie Flexibilität und Erweiterbarkeit der Software und wenig überflüssiger Implementierungsaufwand – erwartet werden.

1.3.1 Code-Reduktion

Ähnliche oder verwandte Routinen haben häufig identische Code-Fragmente zu deren Beginn (z.B. Initialisierungscode) und am Ende (z.B. Finalisierungscode). Mit Hilfe des Prologs und des Epilogs ist es möglich, diese redundanten Code-Fragmente zu eliminieren. die Menge des zu implementierenden Codes wird reduziert ohne dabei den Entwurfsaufwand zu intensivieren. Der korrekte Einsatz von Prolog und Epilog wird durch die Entwicklung eines Systems in Methoden und deren Handlings bereits implizit vorgegeben.

Durch das Aufleiten von Methoden kann das Konzept von Prolog und Epilog auf höheren Abstraktionsebenen fortgesetzt werden (siehe auch Kapitel 1.2.3 Bidirektionale Vererbung).

Ein weiteres Beispiel, wie die Menge von Code mit Hilfe der MOP reduziert wird, ist der Einsatz von Multi-Handlings. Multi-Handlings verhindern die mehrfache Implementierung einer bis auf den zu verarbeitenden Datentyp identischen Routine, natürlich auch unter der Nutzung der Vorteile von Prolog und Epilog.
In der OOP ist dies hingegen nicht immer so einfach möglich. Implementieren beispielsweise verschiedene abgeleitete Klassen eine aus einer gemeinsamen Basisklasse stammende abstrakte oder rein-virtuelle Methode, könnte auch eine weitere Basisklassenschicht für die abgeleiteten Klassen eingefügt werden, in welcher die für die jeweilige Klassengruppe identische Routine implementiert wird.

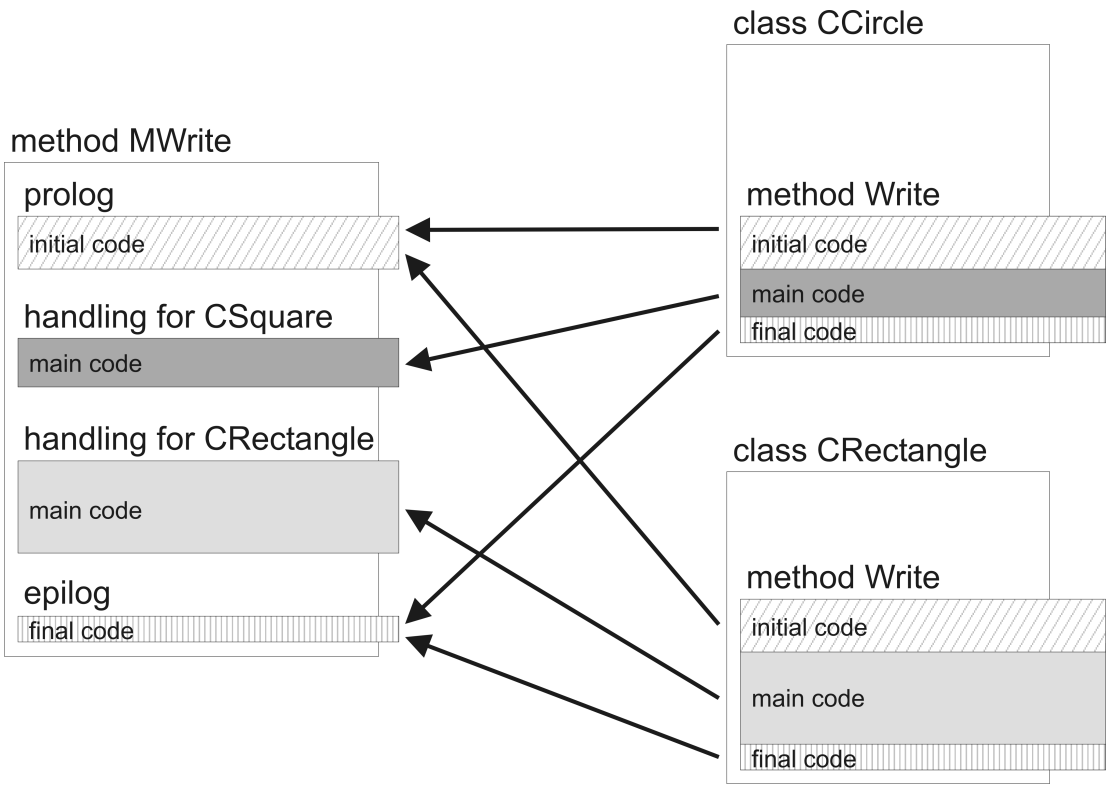


Abb 6: Code-Reduktion mit Hilfe von Prolog und Epilog

Die Klassen ‚CCircle' und ‚CRectangle' implementieren hier beide eine rein-virtuelle Schreibmethode einer Basisklasse. Würde stattdessen eine (MOP-)Methode eingesetzt, wäre das Auftreten der redundanten Initialisierungs- und Finalisierungsfragmente ausgeschlossen.

Dies würde jedoch eine explizite Berücksichtung dieses implementierungsspezifischen Umstands beim Entwurf der Software bedeuten.
Des Weiteren wird die Struktur der Software durch den Einsatz von Multi-Handlings sehr viel flexibler gegenüber Erweiterungen gehalten, ohne dass sie im Laufe ihrer Evolution mehr und mehr an Übersichtlichkeit einbüßen muss. Erweiterungen, wie das Ableiten einer neuen Klasse, müssen so nie größere Veränderungen in der Struktur der Klassenhierarchie nach sich ziehen, um Mehrfachimplementierungen vermeiden zu können. Neue Typen können einfach in die Identifikation eines entsprechenden Handlings aufgenommen werden, um eine bereits implementierte Routine nutzen zu können.

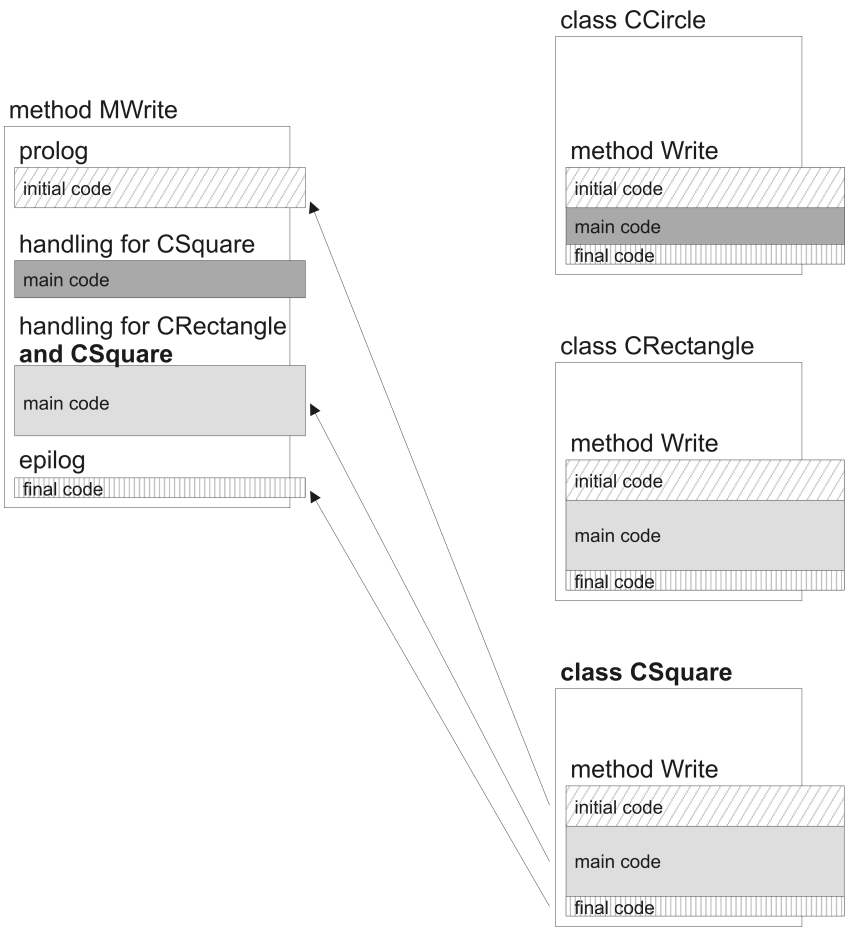


Abb. 7: Code-Reduktion mit Hilfe von Multi-Handlings

Die in diesem Beispiel neu hinzugefügte Klasse ‚CSquare' würde hier die rein-virtuelle Schreibmethode der gemeinsamen Basisklasse identisch implementieren. Das Hinzufügen des neuen Typs in die entsprechende Identifikation eines Handlings der Schreibmethode verhindert dies jedoch auf einfachstem Wege: Die Klassenstruktur bleibt unberührt; das Ergänzen einer weiteren Basisklassen (beispielsweise für Viereckstypen), in welcher die identische Routine implementiert werden könnte, entfällt.

1.3.2 Verbesserte Modularisierung

Auf Grund der Orientierung an den Methoden des umzusetzenden Systems, entsteht durch die MOP eine sehr stark modulare Struktur im gesamten Projekt. Semantisch ähnliche Codeteile sind stets an einer Stelle im Quelltext zentralisiert; identische Codeteile sind eliminiert.

Bespielsweise könnten in einem objektorientiert aufgebauten System verschiedene, einer bestimmten Klassenhierarchie zugehörige Klassen dafür Sorge tragen müssen sich selbst zu visualisieren. In diesem Fall wäre es notwendig jede dieser Klassen mit der Grafik-API zu versorgen. Der Zeichencode wäre dann in die verschiedenen Klassen „eingestreut".
Bei der Verwendung einer Methodenhierarchie, welche all diese Zeichenroutinen

beinhaltet, ist es jedoch möglich, jeglichen Zeichencode in einem Modul zu zentralisieren und redundante Implementierungen (handele es sich um Initialisierungs- oder Finalisierungscode oder auch um die Mehrfachimplementierung einer ganzen Routine für verschiedene Klassen) auszuschließen. Die Einbindung der Grafik-API wäre so auf dieses Zeichenmodul reduziert.

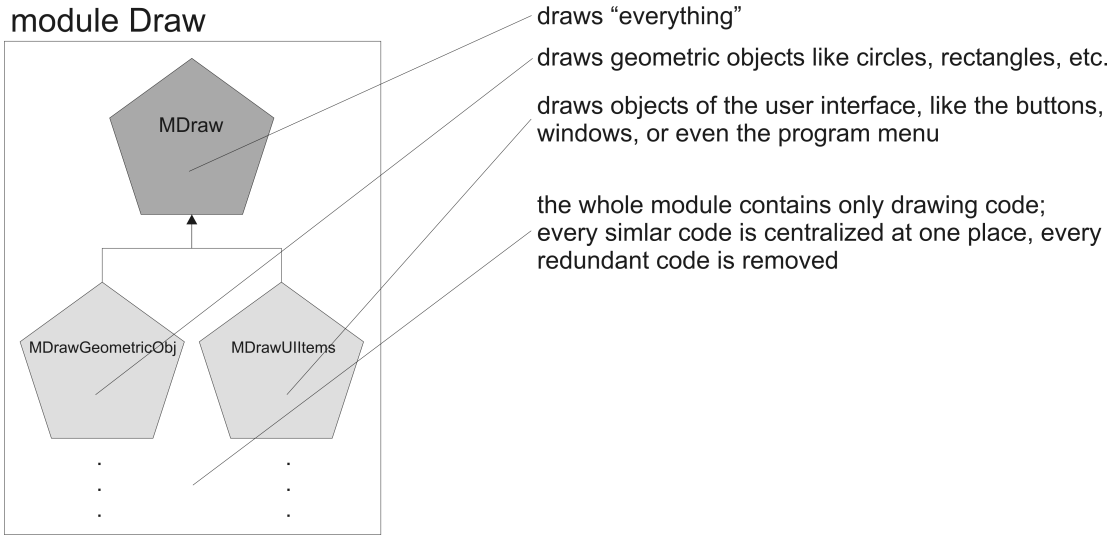


Abb. 8: Modularisierung

Hier übernimmt eine Methodenhierarchie die Implementierung aller Zeichenroutinen. Die Methode ‚MDraw' dient letztlich als Schnittstelle zum Visualisieren jeglicher Objekte des Systems. Die Grafik-API muss ausschließlich in das Modul ‚Draw' eingebunden werden.

1.3.3 Vereinfachung von Programmmodifikationen

Das Modifizieren der Software wird in der MOP besonders durch die redundanzfreie Struktur und die starke Modularisierung der Software erleichtert. Durch die Zentralisierung von semantisch gleichartigem Code und das Nichtvorhandensein redundanter Codeteile können Schlüsselstellen schnell gefunden und Modifikationen zentral an einer Stelle im Quelltext vorgenommen werden.
Das Auftreten von Mehrfachänderungen, wie etwa solche, die für alle Derivate einer bestimmten Basisklasse vorgenommen werden müssen, kann durch Methodenhierarchien und die damit verbundene Staffelung der Prologe und Epiloge der Methoden ausgeschlossen werden. So kann z.B. eine Änderung, welche für einen ganzen Zweig von Methoden übernommen werden soll im Prolog oder Epilog der Integralmethode für diesen Zweig vorgenommen werden.
Des Weiteren ist es möglich mit Hilfe von abstrakten Methoden, durch erneutes Aufleiten deren Integralmethoden, diese bequem durch das Überschreiben bestimmter Handlings anzupassen oder durch das Hinzufügen neuer Handlings zu erweitern, falls die Behandlung neuer Typen erwünscht wird. Auch dabei lassen sich weiterhin die bereits implementierten Codeteile in den Prologen und Epilogen der Methodenhierarchie nutzen.

Durch das Methodenkonzept wird außerdem ein hohes Maß an Flexibilität der Software gegenüber Erweiterungen erreicht. Erweiterungen können vorgenommen werden,

ohne dass redundante Implementierungen entstehen, und ohne dass diese durch eine Umstrukturierung von Teilen der Klassenhierarchien explizit verhindert werden müssen. Eine Erweiterung, wie sie in Abb. 7 in Kapitel 1.3.1 zu sehen ist, bedeutet in der MOP sowohl für die Klassenhierarchie als auch für die Methodenhierarchie lediglich eine Erweiterung, jedoch nie eine Umstrukturierung zum Erhalt der redundanzfreien und sauberen Implementierung. Ein oft zeitaufwändiges und kostenintensives Reengineering im Laufe der Evolution einer Software kann so auf lange Sicht vermieden werden.

2. Die Sprache C-mol

2.1 Einführung in C-mol

Das Konzept der methodenorientierten Softwareentwicklung kann nicht ohne weiteres in konventionellen Sprachen eingesetzt werden. Dafür sind bestimmte methodenorientierte Sprachkonstrukte notwendig – und damit auch ein angepasster Compiler. Die erste methodenorientierte Sprache ist „C-mol“. Wie der Name bereits andeutet, handelt es sich hier um eine Sprache, die von der Programmiersprache C bzw. C++ abstammt.

C ist eine Anfang der siebziger Jahre des letzten Jahrhunderts[1] von Brian W. Kernighan und Dennis W. Ritchie in den Bell Laboratories entwickelte Programmiersprache, die 1989 von ANSI und ISO standardisiert wurde. C wurde Mitte der achtziger Jahre von Bjarne Stroustrup um objektorientierte Konzepte erweitert. In Anlehnung an den Inkrementierungsoperator von C wurde diese Sprache von ihm als C++ bezeichnet[2].
Die neueste Entwicklung ist die von Microsoft entwickelte Sprache C# (gesprochen: „c sharp“) für das .NET-Framework, die in vielerlei Hinsicht stark an Java erinnert. Auch wenn dies von Seiten Microsofts wohl nie offiziell bestätigt wurde, wird dieser Name oft als eine Anspielung auf den „musikalischen Inkrementierungsoperator“ ‚#‘ verstanden, der den angegebenen Ton um einen Halbton erhöht. Im Zuge dieser Tradition steht auch C-mol, dessen Name direkt an den Begriff „Moll“ der Musik erinnert, der für eine bestimmte Art von Akkord steht. Gleichzeitig steht „mol“ aber eben in diesem Fall auch für „method oriented language“.

C-mol ist keine reine methodenorientierte Sprache. Obwohl eine solche „pure“ Sprache konzeptionell möglich wäre, geht C-mol diesen Weg nicht. C-mol verbindet in einer Art Hybridisierung die Konzepte der Methoden- und der Objektorientierung. Genaugenommen ist C-mol ein um das methodenorientierte Konzept erweitertes C++. So kann innerhalb eines Projekts für jede Teilproblemstellung je nach Anforderungen das jeweils beste Paradigma (OOP oder MOP) verwendet werden.

1 nach http://cm.bell-labs.com/cm/cs/who/dmr/chist.html, HOPL II, 1993
2 alle historischen Informationen über C und C++ stammen aus Dirk Louis, C und C++ - Programmierung und Referenz, Kapitel 1.1 bis 1.3

2.2 Sprachkonzepte und Syntax

In den folgenden Unterkapiteln soll nun detaillierter auf die Sprache C-mol eingegangen werden. Dabei sollen syntaktische Mittel, deren Einsatzmöglichkeiten und ihre Relevanz für die methodenorientierte Softwareentwicklung gleichermaßen betrachtet werden.

2.2.1 Methodendeklaration und -definition

Methoden werden syntaktisch weitgehend äquivalent zu Klassen deklariert: Auf das Schlüsselwort `method` folgt der Name der Methode, danach die Parameterliste (Methodenparameter) und dann ein Block mit Handling-Deklarationen oder -Definitionen. Dieser Block muss wie eine Klasse mit einem Semikolon terminiert werden.

Beispiel:

```
method MPrint(int Row)
{
    // Handling-Deklarationen bzw. -Definitionen
};
```

2.2.2 Handlingdeklaration und -definition

Ein Handling wird ähnlich einer OOP-Methode deklariert. Auf den Rückgabetyp folgt die Handling-Identifikation, bestehend aus einem oder mehreren Handling-Datentypen. Um diese eindeutig von dem Rückgabetyp unterscheiden zu können, muss sie in Kleiner-/Größer-Zeichen eingefasst werden. Handlings können entweder direkt innerhalb der Methodendefinition definiert werden oder sie werden an dieser Stelle nur deklariert – dann ist jedoch eine spätere Definition notwendig, bei der dann dem Handling-Typ durch einen Bereichsoperator getrennt der Methodenname vorangestellt werden muss.
Die Rückgabetypen der einzelnen Handlings einer Methode können unterschiedlich sein.
Innerhalb des Handlings kann auf das übergebene Datum über das Schlüsselwort `that` zugegriffen werden. `that` ist immer vom Handling-Datentyp.
Besondere Möglichkeiten im Rahmen der kombinierten Verwendung von Methoden- und Objektorientierung bietet die Tatsache, dass auch Handlings für bestimmte Klassen definiert werden können. Auch die Definition eines Handlings für einen Pointer auf eine Oberklasse und der Aufruf mit der Instanz einer davon abgeleiteten Unterklasse als Input der Methode ist realisierbar.

Beispiel:

```
method MPrint(int Row)
{
    void <int>();  // nur Deklaration, Definition erfolgt extern
    int <char *>()  // interne Definition
    {
        printf("%i: %s\n", Row, that);
        return(strlen(that));
    }
};

void MPrint::<int>()
{
    printf("%i: %i\n", Row, that);
}
```

Besteht die Handling-Identifikation nicht nur aus einem Handling-Datentypen, sondern aus mehreren, durch Kommata getrennt, so spricht man von einem „Multi-Handling“. Dafür können mehrere Typen in den Kleiner-/Größer-Zeichen des Funktionskopfes angegeben werden, die mit Kommata zu trennen sind. Das Handling behandelt dann alle angegebenen Typen. `that` nimmt dabei einen Schnittmengen-Datentyp an (siehe auch Kapitel 1.2.1 Handlings, Prolog und Epilog) und besitzt somit alle Eigenschaften, welche allen Handling-Datentypen in der Handling-Identifikation gemein sind.

Beispiel:

```
void MPrint::<CRationalNumber, CComplexNumber, CRealNumber>()
{
    printf("%i: %s\n", Row, that.GetStringRepresentation());
}
```

2.2.3 Prolog und Epilog

Es können zwei spezielle „Handlings“ in Methoden definiert werden: der Prolog und der Epilog. Diese werden im Zuge eines Methodenaufrufs automatisch vor, respektive nach dem Ausführen eines Handlings aufgerufen, falls sie existieren. Sie werden – ähnlich dem Konstruktor und Destruktor in C++ – mit gleichen Namen wie die Methode und ohne Datentypangabe in Kleiner-/Größer-Zeichen deklariert. Beim Epilog wird dem Methodennamen zusätzlich eine Tilde `~` vorangestellt, um ihn vom Prolog unterscheiden zu können. Ihre Parameterliste muss leer sein. Wie auch Konstruktor und Destruktor in Klassen, haben Prolog und Epilog keinen Rückgabetyp.

Auch in Prolog und Epilog ist ein Zugriff auf `that` möglich. Der Datentyp von `that` nimmt dann den Schnittmengen-Datentyp aller Datentypen für die Handlings in der Methode definiert sind an (siehe auch Kapitel 1.2.1 Handlings, Prolog und Epilog). Es muss also auf alle Typen gleichartig zugegriffen werden können. Dies ist z.B. der Fall, wenn für alle Typen der gleiche Operator überladen ist oder alle Typen ein gleichnamiges Attribut oder eine gleichnamige (OOP-)Methode besitzen.

Beispiel:

```
method MPrint(int Row)
{
    MPrint()  // Prolog
    {
        // Code, der vor jedem Handling ausgeüfhrt wird
    }

    ~MPrint()  // Epilog
    {
        // Code, der nach jedem Handling ausgeüfhrt wird
    }

    // Handlingdeklarationen und -definitionen
};
```

2.2.4 Methodenaufrufe

Aufrufe von Methoden erfolgen über den Namen der Methode, die Angabe ihrer Parameterliste (oder zumindest der leeren Liste: `()`) und darauffolgend einen der beiden Methodenoperatoren ° oder <- (die sogenannten Selektionsoperatoren). Danach folgt der auszuwertende Ausdruck. Dies kann z.B. eine Variable, eine Konstante oder ein Funktionsaufruf sein. Es ist darauf zu achten, dass die Methode ein Handling für den verwendeten Datentyp besitzt, die verwendete Handlingargumentliste zur Signatur des Handlings passt und im Falle der Verwendung des Rückgabewerts auch dessen Typ mit der Deklaration zusammenpasst. Andernfalls wird der Compiler entsprechende Fehlermeldungen ausgeben.
Der Operator <- unterscheidet sich von ° insofern, dass er automatisch eine Dereferenzierung durchführt. Er wird demnach verwendet, wenn nicht ein Pointer auf das gewünschte Datum an die Methode übergeben werden soll, sondern das Datum auf das dieser zeigt. Dies entspricht dem Verhalten von -> und . in C++.
Zu beachten ist, dass die beiden Methodenoperatoren die höchste Priorität unter den Operatoren besitzen. Ein Aufruf wie `MPrint(5)°3+4()` führt also zu einem C-mol-Fehler, denn hier wäre nur 3 der Input für die Methode, wonach eine Argumentliste erwartet werden würde. Zu verwenden wäre hier der Aufruf `MPrint(5)°(3+4)()`.

Beispiel:

```
int EineZahl = 5;
int *pEineZahl = &EineZahl;
char EinText[] = „Hello method-oriented world!“;
int len;
MPrint(1)°EineZahl();        // direkter Aufruf
MPrint(2)<-pEineZahl();      // pEineZahl muss erst dereferenziert
                             // werden
len = MPrint(3)°EinText();   // muss nicht dereferenziert werden,
                             // da das Handling bereits als
                             // Datentyp einen Pointer erwartet
```

Zu beachten ist im Besonderen, dass die Übergabe bei Handlings für nicht-Zeigertypen immer als Referenz geschieht. In der Praxis bedeutet dies, dass that nicht dereferenziert

werden muss und bei einer Veränderung von `that` oder dessen Elementen nicht nur eine lokale Kopie des Datums verändert wird, sondern das Datum im aufrufenden Gültigkeitsbereich. Soll das Datum innerhalb des Handlings nicht veränderbar sein, so sollte eine Deklaration des Handlings mit `const` im Datentyp erfolgen.

Es existieren noch weitere Situationen, in denen eine `const`-Deklaration von Handling-Datentypen sinnvoll sein kann. Zum einen ist dann auch die Übergabe von konstanten Inputs möglich, zum anderen wird dies für die Übergabe ganzer Ausdrücke an die Methode benötigt.

Beispiel:

```
void MPrint2::<const int>()
{
    // ...
}

MPrint2(4)°3();      // auch ein Aufruf mit Konstanten ist möglich
MPrint2(5)°(a+b)(); // auch ein Aufruf mit Ausdrücken ist möglich
```

2.2.5 Vererbung von Methoden

Mit Hilfe der Vererbung können eine oder mehrere Methoden zusammen (Mehrfachvererbung) verändert und erweitert werden.
Die aufgeleitete Methode (Integralmethode) wird deklariert und definiert wie eine konventionelle Methode, nur folgt auf die Methodenparameterliste ein Doppelpunkt und danach eine Auflistung der aufzuleitenden Methoden (Integrationsmethoden), getrennt durch Kommata.

Beispiel:

```
method MAnimiere() : MDrehe, MSchiebe
    // Die beiden Integrationsmethoden auf ein höheres
    // Abstraktionsniveau heben
{
    // ...
};
```

Die entstehende Integralmethode besitzt dann alle Handlings, die auch die Integrationsmethoden besitzen. Prolog und Epilog der Integralmethode werden vor und nach den geerbten Handlings einer Integrationsmethode (inklusive deren Prolog und Epilog) aufgerufen (zur Ausführungsreihenfolge von Prolog und Epilog beim Aufleiten siehe auch Kapitel 1.2.3 Bidirektionale Vererbung; insbesondere Abb. 4a/4b). Des Weiteren werden die Methodenparameter der Integrationsmethoden zu Handlingparametern der einzelnen geerbten Handlings in der Integralmethode. Als Gedankenstütze ist es möglich sich vorzustellen, dass eine Integrationsmethode mit all ihren Handlings zu „einem“ Handling in der Integralmethode wird.
In der Definition der Integralmethode können nun weitere Handlings deklariert und definiert werden. Außerdem ist es auch möglich, bestehende Handlings aus den aufgeleiteten Methoden zu überschreiben, indem hier ein mit der Methoden-

und Handling-Signatur für einen Datentyp in den Integrationsmethoden bereits existierendes Handling erzeugt wird.
Geerbte Handlings können ebenso wie nicht geerbte Handlings überladen werden.

Beispiel:

```
method MDrehe(float Grad)
{
    MDrehe();
    ~MDrehe();
    void <CTisch>();
};

// ...

method MAnimiere() : MDrehe
{
    MAnimiere();   // Prolog wird vor Prolog von MDrehe ausgeführt
    ~MAnimiere();  // Epilog wird nach Epilog von MDrehe
                   // ausgeführt
};

// ...

MDrehe(45)<-pTisch();      // Aufruf des Handlings über die
                           // Integrationsmethode
MAnimiere()<-pTisch(45);   // Aufruf des Handlings über die
                           // Integralmethode; der Methoden-
                           // parameter Grad ist zu einem
                           // Handlingparameter geworden
```

Zu beachten ist noch, dass ein Handling mit einer Identifikation, bestehend aus mehreren Datentypen, für jeden dieser Datentypen einzeln vererbt wird und somit auch Handlings, die Teil dieser Identifikation sind, überschrieben werden können.
Eine Integralmethode wird abstrakt, wenn sie von abstrakten Methoden erbt und nicht alle abstrakten Handlings mit nicht-abstrakten Handlings überschrieben werden. Weitere Informationen dazu finden sich im Kapitel 2.2.6.
Weiterhin wird eine Integralmethode auch polymorph, wenn sie von polymorphen Methoden erbt und nicht alle polymorphen Handlings der Integrationsmethoden mit nicht-polymorphen Handlings überschrieben werden. Weitere Informationen dazu finden sich im Kapitel 2.2.7.

In einigen Fällen ist es beim Aufleiten erwünscht, dass die Methodenparameter der Integrationsmethode nicht zu Handlingparametern der entsprechenden Handlings in der Integralmethode werden, sondern auch in der Integrationsmethode weiterhin als Methodenparameter angesprochen werden können. Dafür ist es notwendig, dass die entsprechenden Parameter inklusive ihrer Datentypen in Klammern hinter die entsprechende Integrationsmethode im Kopf der Integralmethode geschrieben werden.

Beispiel:

```
method MDrawColoredObj(int Color, int x, int y)
{
    void <CRect>();
    // ...
};

// ...

method MDrawAnyObj(int x, int y)
  : MDrawColoredObj(int x, int y)
                // x, y werden als Methodenparameter übernommen,
                // Color wird wie gewohnt zu Handlingparameter
{
    // ...
};

// ...

MDrawColoredObj(COLOR_BLUE, x, y)<-pRect();
    // in der Integrationsmethode existieren nur
    // Methodenparameter

MDrawAnyObj(x, y)<-pRect(COLOR_BLUE);
    // in der Integralmethode sind x und y Methodenparameter
    // geblieben, aber Color ist ein Handlingparameter geworden
```

2.2.6 Abstrakte Methoden und Handlings

Ein Handling ist abstrakt, wenn auf seine Deklaration ein `=0` vor dem abschließenden Semikolon folgt.
Eine Methode ist abstrakt, sobald sie mindestens ein abstraktes Handling besitzt. Wird eine abstrakte Methode aufgeleitet, so wird die Integrationsmethode konkret, wenn alle abstrakten Handlings implementiert werden. Andernfalls ist auch die Integralmethode abstrakt. Wird eine abstrakte Methode aufgerufen, so wird stets die Implementierung des auszuführenden Handlings in der höchsten Aufleitung verwendet.
Somit lassen sich abstrakte Methoden auch als Schnittstellen-Methoden betrachten, da sie (wie auch ihre Äquivalente aus der OOP) eine abstrakte Schnittstelle zur Verfügung stellen, welche es über den Vererbungsmechanismus zu implementieren gilt.
Abstrakte Handlings können auch gleichzeitig virtuell sein, sie werden dann auch als rein-virtuelle Handlings bezeichnet.

Beispiel:

```
method MCallback()
{
    void <int>() = 0;  // MCallback stellt eine Schnittstelle zur
                       // Verfügung, welche nach einem Aufruf
                       // implementiert werden muss
};

//...

int i = 5;
MCallback()°i();       // Führt die Implementierung des Handlings
                       // in der höchsten Aufleitung aus

//...

method MCallbackImplementation() : MCallback
    // Implementiert die zur Verfügung gestellte Schnittstelle
{
    void <int>()
    {
        // Code der bei obigem Aufruf ausgeführt wird
    }
};
```

2.2.7 Polymorphe Methoden und Handlings

Methoden können polymorphe und nicht-polymorphe Handlings zugleich besitzen. Dass ein Handling polymorph sein soll, muss dem Compiler bereits bei der Deklaration mit dem Schlüsselwort `virtual` vor der Deklarationszeile bekanntgegeben werden.
Eine Methode ist polymorph, sobald mindestens eines ihrer Handlings polymorph ist. Polymorphe Handlings einer Methode müssen Handlings für Zeigertypen sein. Auch polymorphe Handlings können überladen werden, jedoch müssen sich alle polymorphen Handlings mit identischen Parameterlisten auch im Typ ihres Rückgabewertes entsprechen. Dies leuchtet bei näherer Betrachtung ein: Soll der Rückgabewert eines polymorphen Handlings verwendet werden, welches durch den Aufruf mit einem zur Entwicklungszeit noch nicht exakt bekannten Typ selektiert wird, so muss zumindest gegeben sein, dass alle in Frage kommenden Handlings den gleichen Rückgabetyp besitzen.
Wird eine polymorphe Methode mit einem Zeiger auf ein Datum aufgerufen, dessen exakter Typ bereits zur Entwicklungszeit bekannt ist, wird keine polymorphe Behandlung durchgeführt, sondern es wird direkt bei der Compilierung ein Handling ausgewählt und später aufgerufen. Dies verhindert unnötige Einbußen des Laufzeitverhaltens. Wird eine polymorphe Methode aber mit einem zur Entwicklungszeit noch nicht exakt bekannten Zeiger aufgerufen (etwa ein Zeiger auf eine Basisklasse), so erfolgt hier bei der Compilierung keine Prüfung (dies wäre auch nicht möglich, da ja nicht sicher festgestellt werden kann, wohin der Zeiger zur Laufzeit zeigen wird). Stattdessen wird Code erzeugt, der zur Laufzeit den Typ des Datums feststellt, daraufhin ein Handling selektiert und dieses dann aufruft. Aus diesem Grund muss für polymorphe Handlings die ANSI C++ Header-Datei „typeinfo“

includiert werden. Stellt sich bei dieser dynamischen Typerkennung heraus, dass der Zeiger auf ein Datum zeigt, für dass kein Handling definiert ist, wird eine Exception mit einem C-mol-Laufzeitfehler als String geworfen.
Aufgrund der Struktur von C++ ist zu beachten, dass polymorphe Handlings nur für Zeigertypen auf polymorphe Klassen funktionieren. D.h. diese Klassen müssen mindestens eine mit `virtual` deklarierte Methode besitzen.

Beispiel:

```
class CStrasse
{
    public:
        virtual ~CStrasse() {}
};

class CFahrzeug
{
    public:
        virtual ~CFahrzeug() {}
};

class CPkw : public CFahrzeug { // ... };
class CBus : public CFahrzeug { // ... };

method MZeichne() // polymorphe Methode zum Visualisieren
                  // polymorpher Objekte verschiedener
                  // Klassenhierarchien, deren Typ erst zur
                  // Laufzeit bekannt ist
{
    virtual void <CStrasse *>();
    virtual void <CPkw *>();
    virtual void <CBus *>();
};
```

Folgender Code könnte dann Fahrzeuge und eine Straße visualisieren:

```
CFahrzeug *pFahrzeug1;
CFahrzeug *pFahrzeug2;
CStrasse *pStrasse;

pFahrzeug1 = new CBus;
pFahrzeug2 = new CPkw;
pStrasse = new CStrasse;

MZeichne()°pFahrzeug1();
MZeichne()°pFahrzeug2();
MZeichne()°pStrasse();
```

2.2.8 Verkettung von Methoden

Mit dem Konzept der Methodenketten ist ein sequenzieller Aufruf mehrerer Methoden mit dem gleichen Datum in einer übersichtlichen Syntax möglich. Alle Methoden inklusive ihrer Methodenparameterlisten werden durch die beiden Selektionsoperatoren (<- oder °) verknüpft und zuletzt mit diesen das entsprechende Datum angehängt.

Sollten sich die Handlingparameterlisten bei den in der Methodenkette verwendeten Handlings unterscheiden, so muss jede Argumentliste einzeln angegeben werden. Zu diesem Zweck ist es möglich, in der Argumentliste nach dem Datum mehrere Listen, durch Semikolon getrennt, anzugeben. Diese werden dann in der umgekehrten Reihenfolge den einzelnen Methoden, bzw. Handlings zugeordnet. Werden weniger Listen angegeben, als Methoden in der Kette verwendet werden, so gilt die letzte Liste für alle noch ausstehenden Methoden. Nach dieser Syntax ist es auch eindeutig, wenn nur eine Liste angegeben wird, die für alle Methoden gilt.

Beispiel:

Angenommen, es existieren folgenden Handlings für einen Quader:

```
class CQuader;

MZeichne::<CQuader>(int xpos, int ypos) { // ... }
MBerechne::<CQuader *>() { // ... }
MInitialisiere::<CQuader>(int Color) { // ... }
```

Dann initiailisiert, berechnet und visualisiert der folgende Code einen Quader:

```
CQuader pQuader = new CQuader;
MZeichne()<-MBerechne()°MInitialisiere()<-pQuader(RED; ; 10, 10);
```

Es auch möglich, eine Methode mehrfach in eine Methodenkette einzuhängen.

Die C-mol-Grammatik in EBNF befindet sich im Anhang B.

3. Der C-mol-Compiler

Grundkenntnisse der Theorie des Compilerbaus werden in diesem Kapitel vorausgesetzt und sind dem Verständnis sehr zuträglich. Einen Überblick über die gewählte Architektur wird der bemühte Leser jedoch auch ohne dieses Wissen erhalten.

3.1 Compiler versus Präprozessor

Bei der Entwickung von Spracherweiterungen stellt sich immer die Frage, ob der Übersetzer in Form eines klassischen Compilers (der Quellcode der Spracherweiterung in Maschinencode umwandelt) oder eines Präprozessors (der Quellcode der Spracherweiterung in eine andere Hochsprache umwandelt) implementiert werden soll. Ein echter Compiler hat den Vorteil einer wesentlich schnelleren Übersetzung und bietet zumeist weitgehendere Möglichkeiten als ein Präprozessor. Ein Präprozessor hat den Vorteil, dass keine Codegenierung und keine Optimierung durchgeführt werden muss, was zu einer wesentlich kürzeren Anfangsentwicklungszeit führt. Weiterhin kann er leicht auf verschiedene Compiler angepasst werden. Damit kann er auf vielen Plattformen verwendet werden und nutzt gleichzeitig noch die Ausgereiftheit der entsprechenden Compiler und Entwicklungsumgebungen.

Der C-mol-Compiler wurde in Form eines Präprozessors entwickelt. Er wurde in der Skriptsprache Python entwickelt, die sich sehr gut für prototyping eignet und, wie in Skriptsprachen üblich, sehr gute Listen- und Stringverarbeitungsfunktionen bietet. Damit war sie perfekt dazu geeignet, für den zunächst als proof-of-concept gedachten C-mol-Compiler verwendet zu werden.
Mittlerweile ist der C-mol-Compiler um viele neue Funktionalitäten erweitert worden. Darüber hinaus wurde auch das Konzept der Methodenorientierten Softwareentwicklung ständig weiterentwickelt und der C-mol-Compiler musste dieser Entwicklung folgen, um eine Möglichkeit der Testbarkeit und der Einsetzbarkeit der Konzepte zu bieten. Damit ist er bereits über den Stand eines proof-of-concepts hinaus und hat den entscheidenden Schritt in Richtung praktischer Anwendbarkeit längst getan.
Momentan wird geprüft, ob eine Implementierung in Form eines Frontends für den Open-Source-Compiler „GNU C++“ möglich ist. In dieser Form würde der C-mol-Compiler dann u.a. kürzere Übersetzungzeiten bieten können, allerdings auch an seiner vielseitigen Anwendbarkeit einbüßen.

3.2 Architektur des Compilers

Die Architektur dieses Präprozessors orientiert sich an einem klassischen Compiler. Viele Präprozessoren beschäftigen sich weder mit lexikalischer, noch mit syntaktischer oder semantischer Analyse (so z.B. der Präprozessor von C++). Sie arbeiten auf einem Niveau, auf dem weder die Syntax noch die Semantik eine Rolle spielt.
Der C-mol-Compiler hingegen besitzt z.B. einen vollwertigen Scanner für alle lexikalischen Elemente von C-mol und C++. Der Parser arbeitet mit einer Recursive-Descent-Architektur, die jedoch nur die C-mol-relevanten Elemente berücksichtigt.
Weiterhin bietet er im Gegensatz zu vielen anderen Präprozessoren eine sehr komfortable Fehlerrückführung, die es ermöglicht, dass jeder Fehler auf die Quellcodedatei und die Zeilennummer des ursprünglichen Codes (d.h. der C-mol-Quellcodedatei) zurückgeführt wird. Die meisten anderen Präprozessoren können Fehler entweder überhaupt nicht zurückliefern, oder geben Fehler nur bezogen auf die von ihnen erzeugten Dateien aus. Damit ist relativ wenig anzufangen, da der Fehler in der Ursprungsdatei begangen wurde und demnach auch nur dort dauerhaft behoben werden kann.
Darüber hinaus liefert der C-mol-Compiler für C-mol-bezogene Fehler sehr klare Fehlermeldungen, mit denen die Fehler leicht aufgespürt und behoben werden können.

Die Vorgänge des C++-Präprozessors und das Linking erledigt der C-mol-Compiler nicht selbst. Er gibt sie jeweils an einen externen Präprozessor und Linker weiter.

Jeder C-mol-Code muss also bis zum Binärcode die folgenden vier Phasen durchlaufen:

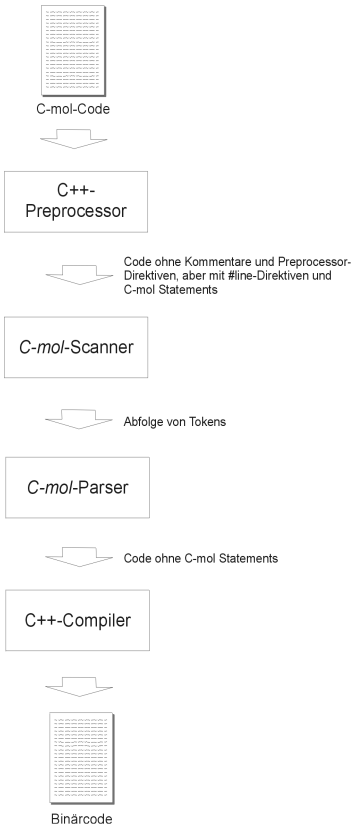


Abb. 9: Ablauf der Übersetzung

Der Code wird zuerst durch den C++-Präprozessor geschickt ...
- Der C++-Präprozessor entfernt alle Kommentare.
- Alle C++-Präprozessor-Befehle werden verarbeitet.
- Die Zeilennummern werden dabei vom C++-Präprozessor erhalten, d.h. er fügt #line-Direktiven ein, die die Zeilennummer dieser Zeile im Ursprungscode angeben.

... dann vom C-mol-Scanner verarbeitet ...
- Dieser zerlegt die Quellcodedatei in ihre Tokens.
- Auf Rückfrage liefert er auch die Zeilennummer, in der das letzte Token aufgetreten ist.

... vom C-mol-Parser umgewandelt ...
- Sämtliche C-mol-spezifischen Sprachelemente werden in äquivalenten strukturierten Code übersetzt.
- Die Zeilennummern der einzelnen Tokens im Ursprungscode werden in einer Tabelle gespeichert (siehe Kapitel 3.5 Das Fehlerrückführungssystem)

... und zuletzt vom externen C++-Compiler und -Linker übersetzt.
- Dieser erstellt aus dem strukturierten bzw. objektorientierten Code Binärcode.

Der C-mol-Compiler fängt die Ausgaben des C++-Compilers ab. Er untersucht jeden Fehler und übersetzt ihn entweder in eine eigene Fehlermeldung – wenn es sich um einen aus der C-mol-Übersetzung entstandenen Fehler handelt – oder liefert ihn an den Benutzer zurück. Dabei überträgt er die Zeilennummern, die der C++-Compiler aus der übersetzten Datei holt, in die Zeilennummern der Ursprungsdatei, damit der Benutzer die zugehörigen Zeilennummern seines eigenen Codes bekommt und auch ohne den Zwischencode die Fehler eindeutig zuordnen kann.

3.3 Scanner

Der Scanner ist das Programmmodul des Compilers, das die lexikalische Analyse durchführt. Der Scanner des C-mol-Compilers enthält den nahezu vollen Funktionsumfang klassischer Scanner. Die Implementation basiert jedoch nicht auf einem Scanner-Generator (wie z.B. lex), sondern basiert direkt auf den mächtigen String- und Listenfunktionen von Python.

Der Scanner zerlegt den Quellcode in seine Grundbestandteile, die sogenannten Tokens. Eine detaillierte Beschreibung hierzu lässt sich beliebiger Literatur zum Thema Compilerbau entnehmen. Weiterhin liefert der Scanner auf Anfrage (des Parsers) die Zeilennummer zurück, in der das aktuelle Token im Quellcode aufgetreten ist. Dies ist für die spätere Fehlerrückführung des Parsers notwendig.

Die Scanner-Klasse stellt folgende Methoden zur Verfügung:

- `ReadCode(Filename, bRunPreprocessor)` initialisiert den Scanner, indem es eine Quellcodedatei einliest. Filename ist der vollständige Dateiname der Quellcodedatei, bRunPreprocessor gibt an, ob der C++-Präprozessor gestartet werden soll.
- `PrescanForPrecompiledHeaders(UntouchedFilename, Outputfile)` sollte nach ReadCode aufgerufen werden und sorgt dann für die korrekte Behandlung der `#pragma c-mol_hdrstop` Direktive. Eine Beschreibung dazu befindet sich im C-mol-Handbuch. Die Routine sucht nach der Direktive und – sollte sie eine finden – kopiert den davor stehenden Code eins-zu-eins in die Ausgabedatei für den C++-Compiler.
- `GetNextSymbol()` liest das nächste Token aus der Datei und liefert ein Tupel (Metatoken, Token, Lexeme) zurück. Das Token enthält eine der definierten Token-Konstanten. Das Lexeme ist die konkrete Ausprägung dieses Tokens. Bei einem Identifier ist dies z.B. der Name des Identifiers. Generell enthält das Lexeme immer genau den extrahierten (relevanten, d.h. ohne Whitespaces, usw.) Code dieses Tokens. Das Metatoken ist eine Art Kategorisierung des Tokens. Hier wird es einer gröberen Struktur zugeordnet, die sich innerhalb des Parsers später als nützlich erweist.
- `GetLastSymbolLine()` liefert die Zeilennummer zurück, in der das zuletzt mit GetNextSymbol bestimmte Token aufgetreten ist.

3.4 Parser

Die Aufgaben des C-mol-Compilers beschränken sich auf das Übersetzen der methodenorientierten Codeteile. Alle nicht methodenorientierten Codeteile werden ohne Veränderung an den C++-Compiler weitergegeben, wobei diesem auch die weitere Fehlererkennung überlassen wird.
Das hat den Vorteil, dass der C-mol-Compiler für viele verschiedene Plattformen und Entwicklungsumgebungen eingesetzt werden kann, da er mit quasi jedem eingesetzen C++-Compiler zusammenarbeiten kann. Damit ist es möglich, die methodenorientierte Softwareentwicklung einer großen Anzahl von Entwicklern und Programmierern in den unterschiedlichsten Bereichen der Softwareentwicklung zugänglich zu machen, ohne auf die Vorzüge der heutzutage sehr ausgereiften C++-Compiler verzichten zu müssen.
So wurden zwar einige leistungsstarke Konzepte der Compilerbau-Theorie zur Entwicklung des Parsers verwendet, jedoch ohne dass seine Struktur streng nach der eines Parsers der klassischen Compilerbau-Theorie ausgerichtet ist.

Der Parser ist in mehreren Klassen implementiert. Dazu gehören die Parser-Klasse selbst, die Getter-Klasse, die Writer-Klasse, die ParseUtils-Klasse sowie die Singleton-Klasse ParseData. Die Parser-Klasse ist dabei die Hauptklasse, welche alle anderen Klassen bedient, jedoch wird im Folgenden auch kurz die Rolle der anderen Klassen beleuchtet.
Abb. 10 soll zunächst veranschaulichen, wie die Methoden der Parser-Klasse einander aufrufen.

Die „Parse-Routinen“, also diejenigen Routinen, die in Abb. 10 dargestellt sind, übersetzen im Prinzip den C-mol-Code direkt (mit den entsprechend notwendigen Modifikationen), also ohne zwischen dem maschinenunabhängigen Frontend (Syntaxbaumgenerierung) und dem maschinenabhängigen Backend (Codegenerierung) zu trennen. Der Vorteil einer konventionellen Parser-Struktur, in welcher eine solche Trennung vorgesehen ist, liegt vor allem darin, dass für eine Portierung von m Sprachen auf n Maschinen m Frontends und n Backends genügen. Ohne eine Frontend-Backend-Architektur würde man für eine solche Portierung normalerweise m*n Compiler entwickeln müssen[1].
Dieser Vorteil wäre im Fall des C-mol-Compilers allerdings nicht sehr stark zu bewerten, da der C-mol-Code in C++ übersetzt wird und somit die Zielmaschine dem angebundenen C++-Compiler überlassen ist. Da der C-mol-Compiler in der plattformunabhängigen Sprache Python entwickelt wurde, kann er somit nicht nur auf jeder Plattform ausgeführt werden, sondern je nach angebundenem C++-Compiler kann auch der generierte Code für jede Plattform compiliert werden.

Wie man aus Abb. 10 ersehen kann, wird als erstes die Routine `ParseCode` aufgerufen. Ihr werden als Argumente eine Quell- und eine Zieldatei übergeben. Die Routine `ParseCode` ruft zunächst den Scanner auf und läuft dann über den (vom Scanner aufbereiteten) Quellcode und sucht gezielt nach methodenorientiertem Code.

[1] nach Niklaus Wirth, Grundlagen und Techniken des Compilerbaus, Seite 3 (Einleitung) und Kapitel 16.5, Seite 142f

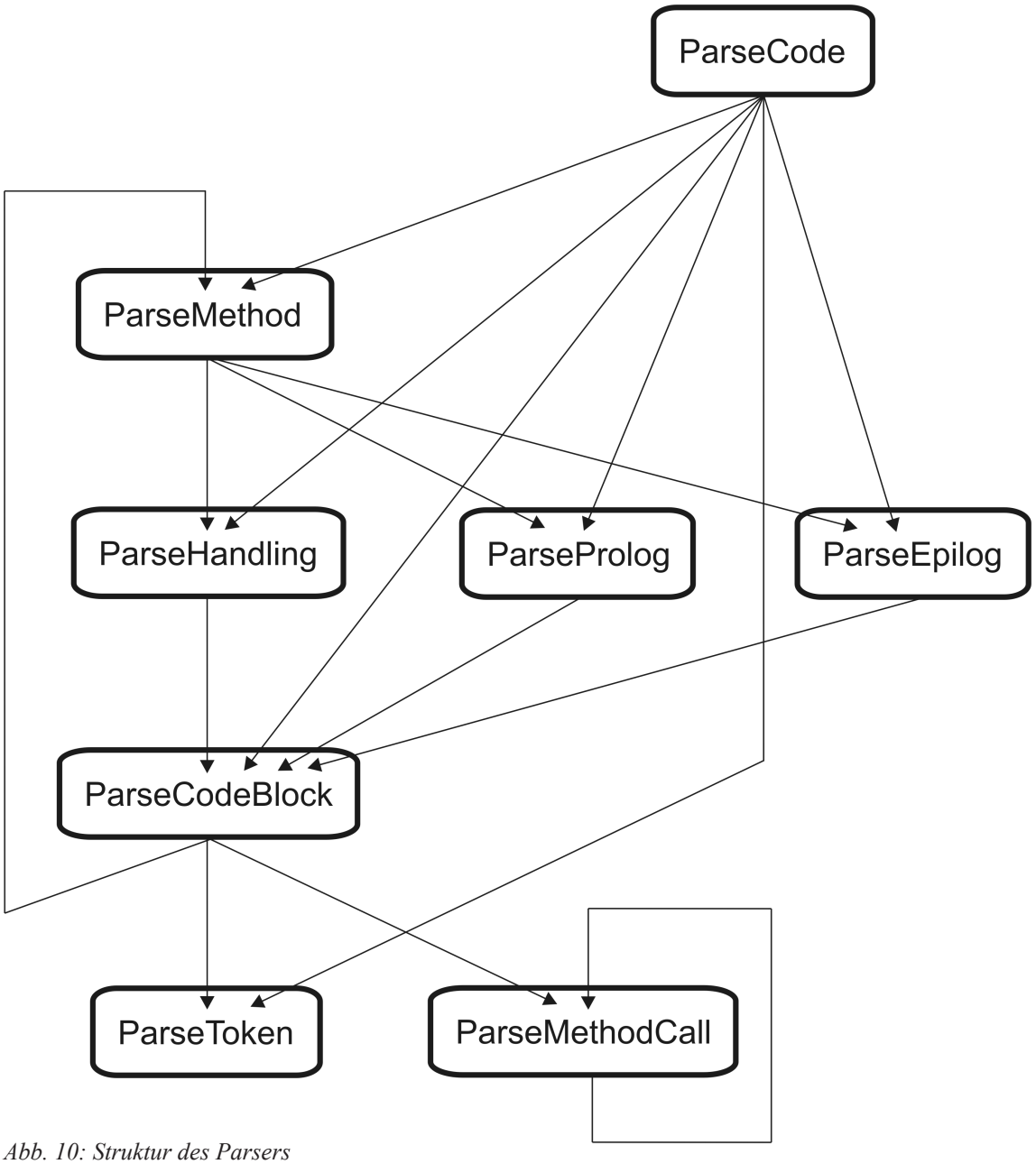


Abb. 10: Struktur des Parsers

Die Methode ‚ParseCode' übersetzt eine vollständige C-mol-/C++-Codedatei. Dabei werden solange alle Tokens ohne Veränderung weitergegeben, bis ein C-mol-Statement gefunden wird. Zum Parsen des nachfolgenden methodenorientierten Codes wird dann die entsprechende Parse-Methode aufgerufen. Dieses Prinzip wird auf den niederen Ebenen entsprechend fortgesetzt.

Dabei werden auch alle Include-Dateien geparst, da diese vom Präprozessor (mit entsprechenden `#line` Direktiven versehen) in die Hauptdatei eingefügt wurden.

Wie die Routinen einander aufrufen, ergibt sich aus der Semantik des Quelltextes: Die `ParseCode` Routine parst die gesamte Quelldatei, die vom Präprozessor erzeugt wurde, daher wird sie auch als erstes aufgerufen und ruft ihrerseits (direkt oder indirekt) alle anderen Routinen auf. In der Quelldatei kann außer Methoden-Aufrufen alles, also methodenorientierter sowie auch anderer Code vorkommen. Methoden-Aufrufe dürfen nur in einem geschlossenen Code-Block auftreten. Es sind auf der Ebene der Quelldatei also einfache „nicht-methodenorientierte Tokens", beginnende

Code-Blöcke, Methoden-Deklarationen und -Definitionen sowie externe Handling- (bzw. Prolog- und Epilog-) Deklarationen und -Definitionen, zulässig. Werden entsprechende Tokens gefunden, die auf solchen Code schließen lassen, werden die dafür zuständigen Parse-Routinen aufgerufen.

Die `ParseMethod` Routine z.B. ruft die Parse-Routinen für Prolog, Epilog und Handlings auf und diese ihrerseits die `ParseCodeBlock` Routine, um den nachfolgenden Code-Block zu parsen.
(Die `ParseHandling` Routine allerdings ruft anstelle von `ParseCodeBlock` eine spezielle `ParseHandlingCodeBlock` Routine auf, die in der obigen Grafik der Übersicht halber nicht aufgeführt ist und auch nicht wichtig für das Verständnis der Struktur des Parsers ist. Der Unterschied zur `ParseCodeBlock` Routine besteht im Wesentlichen darin, dass zusätzliche Funktionsaufrufe für den Prolog (zu Beginn des Code-Blocks) und für den Epilog erzeugt werden. Die Epilog-Aufrufe werden vor jeder Return-Anweisung erzeugt, wobei der Aufruf und die Return-Anweisung als eigener Code-Block geklammert werden - für den Fall, dass die Return-Anweisung in der C/C++ üblichen verkürzten Syntax für Bedingungen und Schleifen eingesetzt wurde. Gegebenenfalls ist es nötig, einen Funktionsaufruf für den Epilog am Ende des Code-Blocks einzufügen, falls sich alle Return-Anweisungen im Handling in einem Bedingungszweig befinden. Dies wird während des Parsens des Handling-Code-Blocks ermittelt.)

Die `ParseCodeBlock` Routine (bzw. `ParseHandlingCodeBlock`) kann wiederum `ParseMethod` aufrufen, da Prolog-, Epilog- und Handling-Deklarationen in Code-Blöcken zulässig sind. Diese Struktur stellt also eine indirekte Rekursion dar. Außerdem schleift sie wie auch die `ParseCode` Routine diejenigen Tokens zum angebundenen C++-Compiler durch, die nicht zu methodenorientiertem Code gehören. Dies erfolgt über die `ParseToken` Routine.

Die `ParseMethodCall` Routine übersetzt einen beliebigen Methoden-Aufruf, also auch Methodenketten. `ParseMethodCall` ist eine rekursive Routine, um auch verschachtelte Methoden-Aufrufe übersetzen zu können (wenn der Rückgabewert eines Methoden-Aufrufs den Input eines anderen darstellt) oder für den Fall, dass in Methoden- oder Handling-Argumentlisten wiederum Methoden-Aufrufe auftreten.

Alle Parse-Routinen übersetzen zwar direkt, allerdings ist der Parser in einigen Fällen auf das Zwischenspeichern von bereits übersetztem Code angewiesen. Daher gibt jede Parse-Routine ein Tupel mit zwei Listen zurück, von denen die erste die Lexeme des zuletzt übersetzten Codes und die zweite die zugehörigen Referenz-Zeilennummern für das Fehlerrückführungssystem enthält. Außerdem kann jede Parse-Routine (über einen optionalen Parameter) in einem „Nur-Rückgabe-Modus“ aufgerufen werden, wobei sie dann den übersetzten Code nur zurückgibt, ohne ihn bereits in die Ausgabedatei zu schreiben.
Bei Handlings für mehrere Handling-Datentypen wird eine Funktion für jeden dieser Datentypen erzeugt, die alle denselben Funktionskörper erhalten müssen. Daher wird der Handling-Code-Block einmal geparst, und das von `ParseHandlingCodeBlock` zurückgelieferte Tupel mit der Liste der geparsten Lexeme und der Liste der Zeilennummern dieses Code-Blocks für alle anderen Handling-Datentypen wiederverwendet.

Ein Beispiel für den Aufruf einer Parse-Routine im „Nur-Rückgabe-Modus“ ist `ParseMethodCall` bei verschachtelten Methoden-Aufrufen, da der Code nicht in der Reihenfolge abgearbeitet werden kann, in der er in die Ausgabedatei geschrieben werden muss und somit der Code höherer Verschachtelungsebenen zwischengespeichert werden muss.

Des Weiteren gibt es neben dem Parser noch einen Getter. Diese Klasse stellt eine Reihe von Routinen zur Verfügung, welche übersetzten Code ausschließlich in einer Liste von Lexemen zurück liefern, anstatt ihn direkt in die Ausgabedatei zu schreiben. Der Getter dient damit Codeteilen, für die es keine direkte äquivalente Umformung gibt. Als Beispiel wäre hier die Routine `GetParameterlist` zu nennen, die eine Liste aller Lexeme, welche beispielsweise die Parameterliste eines Handlings darstellen, zurückliefert. Hier ist es nicht möglich eine solche Parameterliste direkt in die Ausgabedatei zu schreiben, da vorher noch einige Modifikationen wie z.B. das Voranstellen der Methodenparameterliste nötig sind, bevor eine Ausgabe in die Datei erfolgen kann.

Es gibt noch einige weitere Dienstroutinen des Parsers, wie zum Beispiel diverse Konversionsroutinen oder Routinen, welche den Vererbungsmechanismus von C-mol implementieren. Sie sind in der ParseUtils-Klasse implementiert, jedoch nicht essentiell für das Verständnis der Funktionsweise des Parsers, daher soll hier auch nicht weiter auf sie eingegangen werden.

Während des Parse-Vorgangs werden alle wichtigen Daten in entsprechend strukturierter Form im Objekt der ParseData Singleton-Klasse aufgebaut. Es besitzt somit eine Reihe von Listen und assozierten Listen, welche beispielsweise beinhalten, welche Handlings zu welcher Methode gehören, welche Methode von welchen erbt, etc.

Die Writer-Klasse wird vom Parser verwendet, um den übersetzen Code in die Ausgabedatei zu schreiben. Neben der Haupt-Ausgabe-Routine, die letztlich für jeden Code, der geschrieben wird, zuständig ist, ist der Writer wichtig für Codeteile, die in diesem Sinne nicht übersetzt werden müssen, sondern rein generativen Charakter besitzen. Dafür gibt es verschiedene Anwendungen: So müssen zum Beispiel für polymorphe Methoden sogenannte RTTI-Verteilerfunktionen generiert werden, welche dann zur Laufzeit per RTTI (Run Time Type Identification) dynamisch den Typ eingehender Daten ermitteln und in Abhängigkeit davon die Funktionen für die dafür vorgesehenen Handlings aufrufen.
Diese generierten Funktionen lassen sich nicht direkt aus dem Code übersetzen, sondern müssen um die Semantik des C-mol-Codes umsetzen zu können zusätzlich generiert werden. Daher ist zwar der Parser für die Entscheidung ob und an welcher Stelle solche Codeteile generiert werden zuständig, das Generieren an sich bleibt dabei jedoch dem Writer überlassen. Ähnliche Anwendungen des Writers finden sich u.a. auch beim Vererbungsmechanismus.
Die Haupt-Ausgabe-Routine `WriteOutputCode` ist zusätzlich für das Fehlerrückführungssystem von Bedeutung und wird daher auch noch im nächsten Kapitel behandelt.

3.5 Das Fehlerrückführungssystem

Zwar werden im Parser einige (vor allem rein methodenorientierte) Fehler erkannt und per Exception-Handling verwaltet, jedoch ist eine vollständige Fehlererkennung so nicht möglich, da der Parser keinen Syntaxbaum generiert. Eine vollständige Fehlererkennung ist auch nicht nötig, bzw. wäre redundant, da der angebundene C++-Compiler diese ohnehin durchführt. Der Ansatz des C-mol-Compilers besteht also darin, ein System aufzubauen, über das Fehler und Warnungen, die vom angebundenen C++-Compiler erkannt und zurückgegeben werden, auf den C-mol-Code abgebildet werden können.

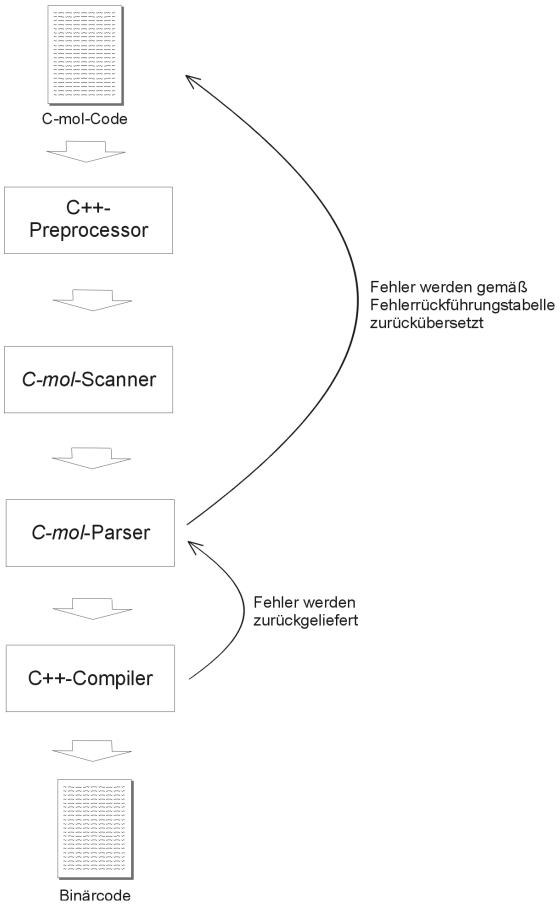


Abb. 11: Fehlerrückführung

Fehler werden vom angebundenen C++-Compiler zurückgegeben und über die vom Parser (bzw. vom Writer als eine seiner Komponenten) aufgebaute Fehlerrückführungstabelle auf den C-mol-Code abgebildet.

Das Fehlerrückführungssystem besteht im Wesentlichen aus einer Fehlerrückführungstabelle, die in Form einer Liste von Tupeln in der Klasse ParseData implementiert ist. Jedes dieser Tupel besteht aus zwei Elementen: erstens dem Namen der Quelldatei in der der Fehler bzw. die Warnung auftrat und zweitens einer Zeilennummer, die die Zeile des Fehlers in der Quelldatei angibt. Während des Übersetzungsvorgangs wird für jede Zeile, die geschrieben wurde, immer ein Eintrag in dieser Liste vorgenommen, wobei als Dateiname der der letzten `#line` Direktive des Präprozessors und als Zeilennummer die verwendet wird, aus der das Token ursprünglich stammte. Auf diese Weise können auch Fehler aus beliebigen Include-Dateien korrekt abgebildet werden. Der Vorgang des Aufbauens dieser Liste findet im Writer in der Methode `WriteOutputCode` statt, da sie am einfachsten (durch „Newline“-Erkennung) feststellen kann, wann eine abgeschlossene Zeile geschrieben wird. Dazu wird Ihr neben einem Tupel, das die Lexeme des zu schreibenden Codes enthält noch eine Liste von Zeilennummern übergeben. Wird ein separates Newline geschrieben, so wird die nächste Zeilennummer aus der Liste geholt und gelöscht, und es wird ein neuer Eintrag in der Fehlerrückführungstabelle mit dieser Zeilennummer vorgenommen.

Gibt der angebundene C++-Compiler später Fehlermeldungen oder Warnungen aus, so wird die Fehlerrückführungstabelle mit deren Zeilennummern indiziert und gibt dann das Tupel mit dem Namen der ursprünglichen Quelldatei und Zeilennummer zurück.

In manchen Fällen ist noch eine Transformation des Textes der Fehlermeldung nötig, da diese nach dem Abbilden auf den C-mol-Code nicht in allen Fällen ganz passend ist. Abb. 11 soll die Funktionsweise des Fehlerrückführungssystems im gesamten Compilierungsvorgang nochmals veranschaulichen.

4. C-mol in der Praxis

4.1 Integration in Entwicklungsumgebungen

Eine Integration von C-mol in Entwicklungsumgebungen ist möglich. Je nach zur Verfügung gestellter Schnittstelle, kann diese Integration mehr oder weniger komfortabel sein. Eine gute Integration konnte für Microsoft Visual Studio 6 bis .NET 2003 erreicht werden. Diese Integration wird im Abschnitt 4.1.1 exemplarisch beschrieben. Plug-ins für die Open-Source Entwicklungsumgebung Eclipse befinden sich in der Entwicklung.
Für andere Entwicklungsumgebungen gilt: Wird eine Integration nicht direkt unterstützt, so ist sie meist doch unter Zuhilfenahme von Makefiles möglich. Weitere Informationen dazu befinden sich im Abschnitt 4.1.2.

4.1.1 Integration in Microsoft Visual Studio

Zur Integration externer Übersetzer bietet Microsoft Visual Studio sogenannte „custom build steps“ an. Diese ermöglichen die manuelle Definition eigener Compilierungsprozesse.

Dafür sind einige Voreinstellungen notwendig: in den Optionen muss Visual Studio der Pfad zum C-mol-Compiler bekannt gegeben werden. Dies lässt sich im Menü unter Tools --> Options --> Directories --> Show directories for: Executable files einstellen. Dort muss der exakte Pfad zum C-mol-Compiler hinzugefügt werden. Ein einfacher Eintrag im Systempfad reicht für Visual Studio nicht aus, da es diesen ignoriert!
Das C-mol-Installationsprogramm für Visual Studio 6 bis .NET 2003 nimmt diese Einstellung automatisch vor. Eine manuelle Änderung ist also nur notwendig, sollte die automatische Konfiguration fehlschlagen.

Weiterhin muss (wie auch im C-mol-Handbuch im Kapitel „Konfigurationsprogramm“ beschrieben) die Konfigurationsdatei für den Microsoft Visual C++-Compiler aktiviert werden. Führt man das Installationsprogramm aus, dass bereits auf Visual Studio vorkonfiguriert ist, muss hier keine Änderung mehr vorgenommen werden.

Nun ist es Visual Studio möglich, den C-mol-Compiler zu finden. Es muss dann für jede Quellcodedatei mit methodenorientierten Inhalten der „custom build step“ aktiviert und einige Einstellungen für den Aufruf des C-mol-Compiler angegeben werden:

```
im Workspace (VC6) bzw. der Solution (VC7/8) -->
im Kontextmenü der C-mol-Datei (Mausklick rechts) -->
unter General -->
   Always use custom build step
und unter Custom Build -->
   Description: Compiling C-mol...
   Commands: C-mol.py $(InputPath)
   Outputs: $(InputDir)\$(InputName).obj
```

Da der Visual C++-Compiler intern vom C-mol-Compiler aufgerufen wird, muss dieser von Visual Studio nicht mehr explizit aufgerufen werden und dies ist bei einem „custom build step“ auch automatisch deaktiviert.
Die Syntaxhervorhebung für die Datei lässt sich über den Eintrag „Properties“ im Kontextmenu des Editorfensters (Mausklick rechts) aktivieren. Dort lässt sich unter „Language“ „C/C++“ auswählen, wobei zu beachten ist, dass so die Schlüsselwörter ‚method‘ und ‚that‘ von der Syntaxhervorhebung ausgeschlossen sind.
Es befindet sich ein Hilfsprogramm in der Entwicklung, das dem Entwickler diese Aufgaben abnimmt, indem es die Anpassung und Verwaltung bestehender Projekte auf Knopfdruck vornimmt.

Dafür, dass nur geänderte Quellcodedateien compiliert werden, sorgt Visual Studio automatisch durch einen Abgleich der Datumsinformationen.
Die Ausgabe von Visual Studio ist grundsätzlich nur die „stdout“ (Kommandozeilenausgabe) des Compilers (Visual C++ oder in diesem Fall der C-mol-Compiler). Da der C-mol-Compiler bei Anbindung an den Visual C++-Compiler die Fehler in der gleichen Syntax zurückliefert wie dieser, sind alle Funktionen von Visual Studio genauso nutzbar (z.B. lässt der Doppelklick auf einen Fehler Visual Studio die betreffende Quellcodedatei öffnen und in die betreffende Zeile springen).

Weitere Informationen und eine exakte Anleitung zur Konfiguration des Visual Studios für den C-mol-Compiler befinden sich im C-mol-Handbuch.

4.1.2 Integration über Makefiles

Sollte eine Entwicklungsumgebung keine direkte Einbindung ermöglichen, ist die Integration des C-mol-Compilers oft über Makefiles möglich. Dies sind Dateien in einem speziellen Format, die den gesamten Erstellungsprozess definieren. Populär geworden sind sie durch Unix und seine Derivate und liegen deshalb noch heute meist im Format des GNU-Make vor. Auch darüber ist der C-mol-Compiler problemlos zu integrieren. Entsprechend dem Verfahren für C++-Quellcodedateien muss dafür für jede Quellcodedatei mit methodenorientierten Inhalten ein Abschnitt definiert werden, der den C-mol-Compiler anstatt dem C++-Compiler aufruft.

Ein solcher Eintrag kann z.B. folgendermaßen aussehen:

```
MPaint.o : MPaint.cmol MDraw.cmol CCircle.cpp CRect.cpp
  @echo „Compiling C-mol: MPaint.cmol...“ && \
  C-mol.py MPaint.cmol
```

D.h. wenn die Datei MPaint.o benötigt wird (also die Objektdatei, die aus dem C-mol-Quellcode erzeugt wird - diese würde z.B. vom Linker benötigt werden), soll ein Hinweistext auf der Eingabezeile ausgegeben werden und dann die Compilierung der Datei MPaint.cmol durch den C-mol-Compiler gestartet werden.
Dabei sagen die Einträge in der ersten Zeile nach dem Doppelpunkt aus, dass die Neuübersetzung auch dann gestartet werden muss, wenn sich die angegebenen Dateien ändern. (MPaint.cmol, weil sich bei einer Änderung des C-mol-Quellcodes natürlich auch der resultierende Objektcode ändert. MDraw.cmol, weil MPaint von MDraw aufgeleitet ist, usw.)

Detailliertere Informationen über das Tool make finden sich z.B. unter http://www.gnu.org/software/make/.

Dies wurde z.B. erfolgreich mit der Integration des C-mol-Compilers in das Falch.net Developer Studio zur Entwicklung für das PalmOS getestet.

Anhang A: Beispielprogramm

```
// Methodenorientiertes C-mol Programm zur Flächenberechnung

#include <stdio.h>
#include <math.h>


#define STYLE_NEWLINE 1
#define STYLE_UNDERLINED 2


struct SRechteck
{
	float SeiteA, SeiteB;
};

struct SKreis
{
	float radius;
};


// Berechnungsmethode
method FlaecheBerechnen()
{

	// Prolog
	FlaecheBerechnen()
	{
		printf("Berechne Flaeche...\n")
	}

	//Handling für Kreis
	float <SKreis>()
	{
		const float PI=3.1415;
		float Flaeche;

		Flaeche = PI*that.radius*that.radius;
		print("%f", Flaeche);

		return Flaeche;
	}

	//Handling für Rechteck
	float <SRechteck>()
	{
		float Flaeche;

		Flaeche = that.SeiteA*that.SeiteB;

		return Flaeche;
	}
};
```

```
// Eingabemethode
method Einlesen()
{

        // Prolog
        Einlesen()
        {
                printf("Bitte geben Sie alle notwendigen Daten fuer ");
        }


        // Handling für Rechteck
        void <SRechteck>()
        {
                printf("das Rechteck ein:\n");
                printf("Seite a: ");
                scanf("%f", &(that.SeiteA));
                printf("Seite b: ");
                scanf("%f", &(that.SeiteB));
                printf("\n");
        }


        // Handling für Kreis
        void <SKreis>()
        {
                printf("den Kreis ein:\n");
                printf("Radius: ");
                scanf("%f", &(that.radius));
                printf("\n");
        }
};



// Ausgabemethode
method Ausgeben(int Style)
{

        // Epilog
        ~Ausgeben()
        {
                if(Style == STYLE_NEWLINE)
                {
                        printf("\n");
                }
                else if(Style == STYLE_UNDERLINED)
                {
                        printf("\n------------------------------\n\n");
                }
        }


        // Handling für Kreis
        void <SKreis>()
        {
                printf("Kreis mit Radius = %f", that.radius);
        }
```

```
        // Handling für Rechteck
        void <SRechteck>()
        {
                printf("Rechteck mit Seite a = %f und Seite b = %f", that.SeiteA, that.SeiteB);
        }


        // Handling für die Flächen-Ausgabe
        void <float>(char *Typ)
        {
                printf("%sflaeche = %f", Typ, that);
        }
};



// Hauptprogramm
int main()
{
        struct SRechteck *pRechteck = new SRechteck;
        struct SKreis *pKreis = new SKreis;
        float Flaeche;

        // Flächenberechnung für Rechteck
        Flaeche = FlaecheBerechnen()<-Einlesen()<-pRechteck();
        Ausgeben(STYLE_NEWLINE)<-pRechteck();
        Ausgeben(STYLE_UNDERLINED)°Flaeche("Rechteck\0");

        // Flächenberechnung für Kreis
        Flaeche = FlaecheBerechnen()<-Einlesen()<-pKreis();
        Ausgeben(STYLE_NEWLINE)<-pKreis();
        Ausgeben(STYLE_UNDERLINED)°Flaeche("Kreis\0");

        delete pRechteck;
        delete pKreis;

        return 0;
}
```

Anhang B: C-mol-Grammatik in EBNF

```
HandlingIdentifikation = {Spezifizierer} Datentyp
          {"," {Spezifizierer} Datentyp} .
```

```
PrologDeklaration = Bezeichner "(" Parameterliste ")" ";" .

EpilogDeklaration = "~" Bezeichner "(" Parameterliste ")" ";" .

HandlingDeklaration = ["virtual"] {Spezifizierer} Datentyp
          "<" HandlingIdentifikation ">" "(" Parameterliste ")"
               ["=" "0"] ";" .
```

```
PrologDefinition = [Bezeichner "::"] Bezeichner "(" ")"
          CodeBlock .

EpilogDefinition = [Bezeichner "::"] "~" Bezeichner "(" ")"
          CodeBlock .

HandlingDefinition = ["virtual"] {Spezifizierer} Datentyp
          [Bezeichner "::"] "<" HandlingIdentifikation ">"
               "(" Parameterliste ")" CodeBlock .
```

```
Methode = "method" Bezeichner "(" Parameterliste ")"
          [":" Bezeichner ["(" Parameterliste ")"]
             {"," Bezeichner ["(" Parameterliste ")"]}]
               "{" [PrologDeklaration | PrologDefinition]
                    {HandlingDeklaration | HandlingDefinition}
                       [EpilogDeklaration | EpilogDefinition] "}" ";" .
```

```
HandlingArgumentliste = [Argumentliste {";" Argumentliste}] .
```

```
MethodenAufruf = Bezeichner "(" Argumentliste ")"
          [{("°" | "<-") Bezeichner "(" Argumentliste ")"}
               ("°" | "<-") Bezeichner "(" HandlingArgumentliste ")"] .
```

Anhang C: Abbildungsverzeichnis / Quellenangaben

Abbildungsverzeichnis

Abb. 1: Typischer Entwicklungsprozess vom realen Problem zur Implementierung
(S. 8)

Abb. 2: Eine einfache Methode
(S. 10)

Abb. 3: Beispiel für erweiterten Polymorphismus
(S. 11)

Abb. 4a: Methodenvererbung (UML-ähnliche Darstellung)
(S. 12)

Abb. 4b: Staffelung der Prologe und Epiloge bei der Vererbung von Methoden
(S. 12)

Abb. 5: Verkettung von Methoden
(S. 13)

Abb. 6: Code-Reduktion mit Hilfe von Prolog und Epilog
(S. 15)

Abb. 7: Code-Reduktion mit Hilfe von Multi-Handlings
(S. 16)

Abb. 8: Modularisierung
(S. 17)

Abb. 9: Ablauf der Übersetzung
(S. 32)

Abb. 10: Struktur des Parsers
(S. 36)

Abb. 11: Fehlerrückführung
(S. 39)

Alle Abbildungen wurden von den Autoren selbst erstellt.

Quellenangaben

Louis, Dirk; C und C++ - Programmierung und Referenz; München: Markt und Technik, Buch- und Software-Verlag, 1998.

Wirth, Niklaus: Grundlagen und Techniken des Compilerbaus; Bonn, Paris u.a.: Addison-Wesley, 1996.

Webseite der Bell Labs Computing and Mathematical Science Research Division; verwendeter Link: http://cm.bell-labs.com/cm/cs/who/dmr/chist.html, Stand 2001.

Webseite des GNU Operating System - Free Software Foundation (FSF); verwendeter Link: http://www.gnu.org/software/make/, Stand 2004