Meta-Konzepte in objektorientierten Sprachen - unter besonderer Berücksichtigung von C++

Diplomarbeit

vorgelegt von
Wolf Siberski

Fachbereich Informatik
der Universität Hamburg,
Arbeitsbereich Softwaretechnik

Juni 1995

Hiermit versichere ich, daß ich diese Arbeit selbständig angefertigt habe. Dabei habe ich keine außer den angegebenen Quellen und Hilfsmitteln verwendet.

Hannover, den 28.6.1995

Inhaltsverzeichnis

1 Einführung

In objektorientierten Programmiersprachen werden Objekte dazu verwendet, Gegenstände eines Anwendungsbereichs zu modellieren. Nun spricht nichts dagegen - und es ist auch oft sinnvoll - als Anwendungsbereich ein objektorientiertes Programm selbst zu wählen. Dann sind die „Gegenstände des Anwendungsbereichs“ Konstrukte der Programmiersprache, in der das Programm geschrieben ist. Objekte, die diese Konstrukte modellieren, werden Metaobjekte genannt (zur Bezeichnung „Meta“ s. 2.3). Wie jedes Objekt sind auch Metaobjekte Exemplare von Klassen, den sog. Metaklassen. Die Menge der (öffentlichen) Schnittstellen dieser Metaklassen bildet ein Metaobjekt-Protokoll (MOP).

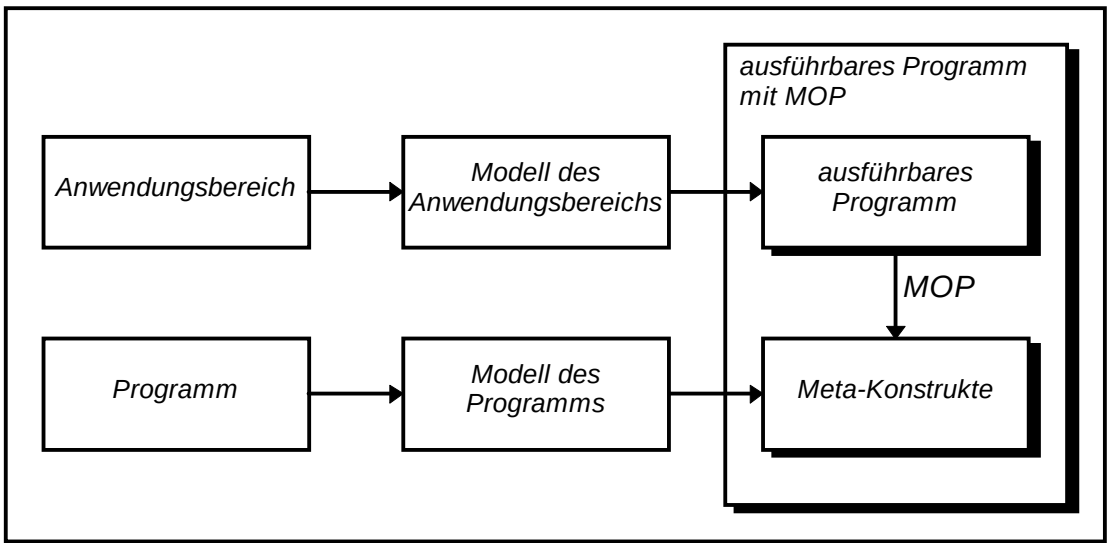

Diagramm 1.1 Modellierung eines Anwendungsbereches und Modellierung eines Programms

Meta-Information wird aber auch in objektorientierten Sprachen nicht nur objektorientiert modelliert, sondern zum Teil auch prozedur- oder datenorientiert. Diese Arbeit beschäftigt sich daher nicht nur mit MOPs im engen Sinne, sondern mit allen Konstrukten, die den Zugriff auf Information[1] über ein Programm innerhalb des Programmes selbst erlauben; solche Konstrukte seien im folgenden Meta-Konzepte genannt.

In Sprachen mit wenig Typ-Information sind Meta-Konzepte oft automatisch vorhanden. So sind z.B. in LISP sowohl Funktionen als auch Daten Listen, die mit den normalen Listenzugriffs- und Modifikationsfunktionen bearbeitet werden können. In getypten Sprachen geht solche Information jedoch zur Laufzeit verloren; das liegt daran, daß der Zugriff auf ein Konstrukt als etwas anderes (im Beispiel der Zugriff auf eine Funktion als Liste) durch die Typisierung ja gerade verhindert werden soll ([Cardelli89], S. 2). Eine getypte Sprache muß daher explizite Sprachkonstrukte zur Verfügung stellen, wenn das Programm Zugriff auf sich selbst erhalten soll. Auch in ungetypten und nur schwach getypten Sprachen kann es sinnvoll sein, ein MOP einzuführen, damit der Zugriff auf die Repräsentation über eine wohldefinierte Schnittstelle erfolgt; dies fördert die Verständlichkeit und die Portabilität der Programme.

Diese Arbeit ist aus der Beschäftigung mit MOPs für C++ entstanden und legt daher einen Schwerpunkt auf diese Sprache. Im zukünftigen ANSI/ISO C++ wird es nur ein rudimentäres MOP geben (s. 6.3), dessen Funktionalität z.B. für Frameworks nicht ausreicht. Daher sind schon viele MOPs für C++ entwickelt worden, die entweder mit dem Standard-Preprozessor oder von eigens dafür entwickelten Precompilern generiert werden. Jedes dieser Protokolle bietet aus gutem Grund nur einen Teil aller möglichen MOP-Funktionalität. Denn erstens wäre ein MOP, das die für alle denkbaren Anwendungen benötigte Funktionalität bieten würde, viel zu umfang-

[1] Unter „Zugriff auf Information“ wird hier und im folgenden sowohl der lesende als auch der verändernde Zugriff verstanden.

reich; die meisten Anwendungen würden nur einen kleinen Teil eines solchen MOPs verwenden. Dies widerspricht jedoch dem Grundsatz von C++, „What you don't use, you don't pay for (zero-overhead rule)" ([Stroustrup94], S. 120). Zweitens steht überhaupt noch nicht fest, wie ein ideales MOP für C++ aussehen müßte. Die bisherigen Erfahrungen (s. Kap. 6) deuten darauf hin, daß je nach dem Schwerpunkt der Anwendung unterschiedliche Ansätze optimal sind.
Für das Problem des zu großen Umfangs bieten diejenigen MOP-Generatoren eine Lösung, bei denen der Entwickler festlegen kann, welche Bestandteile aus einer vorgegebenen Liste an MOP-Funktionalitäten für welche Klasse erzeugt werden können. Aber auch mit diesen Generatoren läßt sich nur für bestimmte vorgegebene Aufgabenstellungen (z.B. Skriptsprachen-Anbindung, Verteilte Objekte) Code erzeugen. Benötigt ein Anwendungsentwickler etwas ausgefallenere MOP-Funktionalität, kommt er mit diesen Generatoren nicht weiter, es sei denn, er ist gleichzeitig der Entwickler des MOP-Generators.
Die Lösung dieses Problems liegt darin, dem Entwickler die Meta-Information nicht erst zur Laufzeit, sondern schon zur Übersetzungszeit zur Verfügung zu stellen. Dann liegt die Kontrolle darüber, was für ein Laufzeit-MOP der MOP-Generator erzeugen soll, vollständig beim Entwickler. Dadurch kann er es genau auf die Bedürfnisse seiner Anwendungen zuschneiden.
Der im Rahmen dieser Arbeit entwickelte Precompiler MopGen realisiert diesen Ansatz. Der Anwendungsentwickler kann mit Hilfe einer eigenen Meta-Sprache MopSpec sein Laufzeit-MOP selbst spezifizieren. Auf diese Weise läßt sich applikationsspezifische Laufzeit-Meta-Information erzeugen, ohne einen eigenen MOP-Generator entwickeln bzw. einen vorhandenen modifizieren zu müssen.

1.1 Aufbau der Arbeit

Der erste Teil der Arbeit beschäftigt sich mit Metaobjekt-Protokollen im allgemeinen. Kapitel 2 soll kurz in die verwendete Begrifflichkeit einführen. Den Kern des ersten Teils bildet Kapitel 3, das den softwaretechnischen Nutzen von Meta-Konzepten aufzeigt. In Kapitel 4 wird beschrieben, welche Entwurfsentscheidungen bei der Entwicklung eines MOP möglich sind. Kapitel 5 bietet eine Beschreibung der MOPs ausgewählter OO-Sprachen. In Kapitel 6 werden einige schon realisierte MOPs für C++ vorgestellt. Anhang B enthält eine skizzenhafte Beschreibung, wie man diese MOPs mit MopGen generieren kann.
Der zweite Teil beschäftigt sich mit dem vom Autor realisierten Metaobjekt-Protokoll-Generator MopGen. Nach einer kurzen Einführung (Kap. 7) wird die Meta-Sprache MopSpec vorgestellt, in der sich die eigene MOP-Funktionalität spezifizieren läßt (Kap. 8). Die Beschreibung der technischen Realisierung von MopGen in Kapitel 9 kann der Nicht-Interessierte ohne weiteres überspringen. In Kapitel 10 wird das Beispiel-MOP beschrieben, das zusammen mit MopGen entwickelt wurde und einen Belege für die Praxistauglichkeit dieses Ansatzes liefert. Es kann auch als Ausgangspunkt für den Entwurf eines eigenen MOP dienen. Der MopSpec-Quelltext des Beispiel-MOPs ist in Anhang A zu finden.
Den Abschluß bilden ein Ausblick auf die mögliche Weiterentwicklung von MopGen und die Zusammenfassung (Kapitel 11).

I Meta-Konzepte

2 Grundbegriffe der Objektorientierung

Da es verschiedene Auffassungen darüber gibt, was Begriffe wie Objekt, Klasse, Methode, usw. bedeuten, ist es angebracht, kurz anzudeuten wie diese Begriffe in dieser Arbeit verwendet werden.

Objekte haben einen Zustand und eine Menge von Zugriffsoperationen, mit denen dieser Zustand gelesen oder modifiziert werden kann. Die Art und Weise, auf die der Zustand gespeichert wird, heißt **Repräsentation**. Diese Repräsentation besteht aus einer Menge von Variablen, den sog. **Exemplarvariablen**. Die Zugriffsoperationen heißen **Methoden**. Die Menge aller Methoden eines Objekts wird auch **Schnittstelle** des Objekts genannt. Eine **Klasse** ist die explizite Beschreibung der Repräsentation und der Schnittstelle einer Art von Objekten. Die Objekte einer Klasse heißen **Exemplare** dieser Klasse. Ein Objekt ist immer Exemplar genau einer Klasse. Von einer Klasse können **Subklassen** abgeleitet werden. Eine solche Subklasse übernimmt zunächst die Beschreibung ihrer **Superklasse**; diese Beschreibung kann je nachdem, was die verwendete Programmiersprache erlaubt, in mehr oder weniger engen Grenzen modifiziert und ergänzt werden, z.B. durch Hinzufügen von Exemplarvariablen oder **Überladen** von Methoden, d.h. Angeben neuer Implementationen für die Methoden.

2.1 Die Beziehung zwischen Typen und Klassen[1]

Eine Superklassen/Subklassen-Beziehung muß immer explizit vom Programmierer angegeben werden. Ein **Typ** dagegen ist charakterisiert durch eine Menge von Prädikaten (Eigenschaften). Ein Objekt x hat den Typ T genau dann, wenn x die Eigenschaften erfüllt, durch die T charakterisiert ist. Die Prädikate können im Prinzip beliebige Aussagen über Objekte sein. Folgende Prädikate sind üblich:

- Aussagen über die Klasse eines Objekts
- Aussagen über die Namen der Methoden eines Objekts
- Aussagen über die Schnittstelle eines Objekts, also nicht nur die Namen, sondern auch die Argumente der Methoden
- Aussagen über das Verhalten der Methoden (Vor- und Nachbedingungen, Invarianten).

Ob ein Objekt von einem bestimmten Typ ist, wird immer automatisch vom Compiler oder zur Laufzeit berechnet (type inference). Daraus ergibt sich, daß eine Supertyp-/Subtyp-Beziehung ebenfalls nicht explizit deklarierbar ist, sondern sich implizit über die Typberechnung ergibt:
Soweit die hier vorgestellten Sprachen getypt sind, wird der Typ eines Objekts über seine Klassenzugehörigkeit definiert. Folgerichtig wird für die Subtyp-Berechnung immer die Subklas-

Typ charakterisiert durch Aussagen über	**Subtyp-Eigenschaft ergibt sich durch**
Klassen-/Subklassen-Beziehung	Subklassen-Eigenschaft
Methoden-Namen	Mehr Methoden
Schnittstelle	Schnittstellen-Kompatibilität
Verhalten	Schwächere Vorbedingungen Stärkere Nachbedingungen Mehr Invarianten

sen-Eigenschaft herangezogen, d.h. ein Objekt ist genau dann vom Typ T, wenn es ein Exemplar der Klasse T oder einer Subklasse von T ist. Aus diesem Grund gibt es in den untersuchten

[1] Dieser Abschnitt stützt sich auf [PalSch91] und [Wegner87].

Sprachen auch keine Typ-Meta-Information im eigentlichen Sinne, sondern nur Klassen-Meta-Information.

2.2 Die Terminologie der verschiedenen OO-Sprachen

In den verschiedenen OO-Sprachen hat sich (leider) eine unterschiedliche Terminologie eingebürgert. Teils sind die unterschiedlichen Begriffe auch in Unterschieden der Sprachen begründet (generic functions in CLOS sind keine Methoden im üblichen Sinne), teils scheint es jedoch, als werde nur der Abgrenzung halber eine neue Begrifflichkeit eingeführt (s. in der folgenden Tabelle die Zeile Exemplarvariable, in der das in allen Sprachen identische Konzept jedesmal anders genannt wird). Die daraus resultierende Problematik ließ sich nicht vollständig lösen. Es wurde folgender Kompromiß gewählt: In denjenigen Teilen der Arbeit, die allgemein gehalten sind, werden üblicherweise die in der letzten Spalte angegebenen Begriffe verwendet; in den Teilen, die eine bestimmte Sprache behandeln, wird üblicherweise die sprachspezifische Terminologie verwendet[1].

C++	CLOS	Eiffel	Smalltalk	In dieser Arbeit gebrauchter Begriff
object	**object**	**object**	**object**	**Objekt**
class	**class**	**class**	**class**	**Klasse**
object of class	**instance**	**instance**	**instance**	**Exemplar**
data member	**slot**	**feature:attribute**	**instance variable**	**Exemplarvariable**
static data member			**class variable**	**Klassenvariable**
(virtual) member function	**generic function method**	**feature: routine**	**method**	**Methode**
static member function	**-**		**class methods**	**Klassenmethode**
this	**- (erster Param.)**	**Current**	**self**	**-**
base class	**direct superclass**	**parent (class)**	**superclass**	**Superklasse**
derived class	**subclass**	**heir (class)**	**subclass**	**Subklasse, abgeleitete Klasse**

Tabelle 2.1 Die OO-Terminologie in den unterschiedlichen Sprachen

2.3 Exkurs: „Meta“ - eine kurze Begriffsgeschichte

Die Vorsilbe Meta kommt vom griechischen μετα, das „nach“ bedeutet. Der Ursprung ihrer heutigen Bedeutung liegt in einem Teil der Schriften Aristoteles, die er selbst als „Erste Philosophie“ (πρωτη φιλοσοφια) bezeichnete. Die Schriften Aristoteles’ wurden lange nach seinem Tod von einem Redaktor neu zusammengestellt. Dieser Redaktor ordnete die erste Philosophie hinter den Schriften über die Physik ein und gab ihr den zweideutigen Titel „Μετα τα φυσικα“ („nach der Physik“). Damit ist zunächst einfach bloß die bibliothekarische Ordnung der Schriften gemeint (nach den Schriften über die Physik); der Titel wurde dann aber auch bald so verstanden, daß in der Metaphysik von den Dingen „hinter“ den physikalischen Gegenständen die Rede sei.
Der nächste Schritt in Richtung unseres Begriffs geschieht erst zweitausend Jahre später, gegen Ende des 19. Jahrhunderts. Damals war die Metaphysik als „reine Spekulation“ verrufen,

[1] Ausnahmen bestätigen die Regel.

während die Naturwissenschaften, insbesondere die Physik, eine Vorbildrolle als „objektive Wissenschaften“ spielten. Zu dieser Zeit entstand im Zusammenhang der Entwicklung der nicht-euklidischen und mehr als dreidimensionalen Geometrie in der Mathematik eine Diskussion, ob es nicht auch in Wirklichkeit mehr als drei (Raum-)Dimensionen gebe. Diese Überlegungen wurden, vor allem wegen der unreflektierten Vermischung mathematischer und philosophischer Fragestellungen, von den Vertretern der euklidischen Geometrie und des dreidimensionalen Raumes mit dem abwertend gemeinten Begriff „Metamathematik“ bedacht, in Analogie zum Begriffspaar Physik/Metaphysik. Die Vertreter der nicht-euklidischen Geometrie taten das wohl Geschickteste: sie nahmen die Bezeichnung als Ehrentitel, wie z.B., Helmholtz:

> „Der Name ist allerdings in ironischem Sinne von Gegnern gegeben, nachgebildet der Metaphysik. Da aber die Bearbeiter der Nicht-Euklidischen Geometrie deren objective Wahrheit nie behauptet haben, so können sie den Namen sehr wohl acceptiren.“

Die Entdeckung der nicht-euklidischen Geometrie führte in der Mathematik zur Einführung der sog. axiomatischen Methode. Die Mathematik sollte nicht mehr von in irgendeiner Weise „evidenten“ oder „offensichtlichen“ Axiomen (wie in der euklidischen Geometrie) ausgehen, sondern die Axiome durften jetzt beliebig gewählt werden. Diese „neue“ Mathematik wurde besonders von Hilbert vorangetrieben. Es entstanden neue Probleme, wie z.B. die Frage der Widerspruchsfreiheit von Axiomensystemen. Der Begriff „Metamathematik“ erhielt eine neue Bedeutung: das Teilgebiet der Mathematik, das beliebige axiomatische Theorien der Mathematik zum Gegenstand hat. In diesem Teilgebiet wurden die ersten Theorien der formalen Sprachen entwickelt und Probleme wie die Widerspruchsfreiheit oder Vollständigkeit mathematischer Theorien behandelt.

Der Begriff Metasprache stammt aus einem Teilgebiet der Metamathematik, der formalen Semantik. Schon die ersten Theorien der formalen Semantik (Tarski, Carnap) trafen eine Unterscheidung zwischen Objektsprache und Metasprache. Während in der Objektsprache (nur) Aussagen über Gegenstände gemacht werden können (z.B. „Raben sind weiß“), kann man in der Metasprache Aussagen über die Objektsprache machen (z.B. „Der Satz ‘Raben sind weiß’ ist grammatisch korrekt“). Objektsprache und Metasprache dürfen nicht identisch sein, weil sonst Antinomien auftreten (z.B. „Dieser Satz ist falsch“).[1]

An den Begriff Metasprache schließt der Begriff Meta-Konzept in Programmiersprachen direkt an. Programmierkonzepte dienen dazu, einen Gegenstandsbereich zu modellieren, Meta-Konzepte modellieren die Programme. Die objektorientierten Meta-Konzepte kommen - wie nicht anders zu erwarten - aus dem Bereich der Entwicklung objektorientierter Sprachen: Der Begriff Metaklasse wurde zuerst im Smalltalk-Umfeld geprägt, der Begriff Metaobjekt-Protokoll im CLOS-Umfeld.

[1] Siehe [HWPh80], Artikel „Metaphysik“, „Metalogik“, „Metamathematik“ und „Metasprache/Objektsprache“

3 Wozu dienen und was leisten Metaobjekt-Protokolle?

Die Anwendungsgebiete und die Bandbreite der Funktionalität von Metaobjekt-Protokollen sind so groß, daß es schwer fällt, einen gemeinsamen Nenner zu finden. Eines scheint jedoch allen Features gemeinsam: Sie stellen dem Entwickler Information und damit Eingriffsmöglichkeiten wieder zur Verfügung, die zuvor absichtlich von ihm ferngehalten wurden.
Dreierlei Art von Information läßt sich hierbei identifizieren:

- Information über das Programm (z.B: über die Methoden einer Klasse). Sie dient zum Überschreiten der Grenzen streng getypter Sprachen (bzw. zur Einführung einer einheitlichen Schnittstelle bei schwach getypten Sprachen).
- Information über die Sprachimplementation (z.B. über die Größe eines Objekts). Sie ermöglicht dem Entwickler Optimierungen und teilweise die Einführung neuer Funktionalität.
- Information über den Interpreter. Mit Hilfe dieser Eingriffsmöglichkeit kann der Entwickler selbstmodifizierende Programme schreiben und die Sprache selbst verändern.

Bevor diese drei Informationsarten näher erläutert werden, soll zunächst ein einfaches Anwendungsprogramm veranschaulichen, an welchen Stellen ein Metaobjekt-Protokoll softwaretechnisch sinnvoll eingesetzt werden kann und sollte.

3.1 Ein Anwendungsbeispiel

3.1.1 Die „Systemvision“

Als Beispielanwendung dient eine Mini-Dokumentverwaltung, die zur Verwaltung und Bearbeitung von einfachen Dokumenten dienen soll. Solche Dokumente sind z.B. Briefe, Faxe, e-mail, Notizen, usw. Es gibt Körbe, in denen die Dokumente abgelegt und aus denen sie herausgenommen werden können. Jeder Korb kann beliebig viele Dokumente beliebiger Art enthalten.
Um die Dokumente angemessen bearbeiten zu können, gibt es verschiedene Editoren, z.B. einen Brief-Editor, einen Fax-Editor, usw. Das Auswählen eines Dokuments zur Bearbeitung bewirkt, daß derjenige Editor gestartet wird, mit dem sich das Dokument am besten bearbeiten läßt. Für unbekannte Dokumenttypen gibt es einen allgemeinen Editor. Der gestartete Editor zeigt das ausgewählte Dokument zur Bearbeitung an. Eine Folge von Bearbeitungsschritten kann als Makro aufgezeichnet und später wiederholt werden.
Jedes Dokument hat einen Titel, ein Erzeugungsdatum und ein Datum der letzten Bearbeitung. Diese Attribute dienen den Körben dazu, die Dokumente zu ordnen und als Listen anzuzeigen. Spezielle Dokumente können beliebige andere Attribute haben, z.B. könnte in einem Fax die Telefonnummer des Absenders enthalten sein. Natürlich gehen die Dokumente nicht bei Programmende verloren; alle Körbe und alle Dokumente werden gespeichert, sodaß sie beim nächsten Programmstart wieder geladen werden können.

3.1.2 Die Realisierung mit Hilfe eines Metaobjekt-Protokolls

An dieser Stelle soll ein fragmentarischer Entwurf des Anwendungsbeispiels skizziert werden. Das Ziel ist, die Komponenten Korb, Dokument und Editor möglichst lose zu koppeln, d.h. sie sollen möglichst wenig „Wissen“ über die anderen Komponenten haben. Außerdem soll ein möglichst großer Teil der im vorigen Abschnitt beschriebenen Funktionalität mit Hilfe eines Metaobjekt-Protokolls realisiert werden.

Alle Dokumente sind von der abstrakten Basisklasse `Dokument` abgeleitet, die als Schnittstelle die Methoden `GibTitel()`, `GibErzeugDatum()` und `GibBearbDatum()` hat. Jede davon abgeleitete Dokumentklasse hat weitere Methoden, die das Anzeigen und Bearbeiten erlauben.
Ebenso sind alle Editoren von der Basisklasse `Editor` abgeleitet. Sie bietet die Methoden `Open( Document* )` und `Close()`.

Ein Korb ist ein Dokumentbehälter. Um die Dokumente anzeigen zu können, muß er etwas Wissen über sie haben. Es genügt jedoch, wenn er die Schnittstelle der Klasse `Dokument` benutzt; er weiß also nichts über die speziellen Dokumenttypen. Dadurch ist es möglich, neue Dokumenttypen einzuführen, ohne etwas an der Spezifikation und Implementation für Körbe ändern zu müssen.
Beim Auswählen eines Dokuments soll der passende Editor gestartet werden. Um diese Funktionalität bieten zu können muß der Korb auch etwas über Editoren wissen. Es genügt, daß er die Basisklasse aller Editoren kennt. Mit Hilfe der späten Erzeugung (s. 10.3) kann er ein zu einem ausgewählten Dokument passenden Editor erzeugen. Anschließend ruft er die `Open()`-Methode dieses Editors auf. Durch dieses Vorgehen wird erreicht, daß Dokumente keinerlei Wissen über Editoren haben.
Ein Editor für eine spezielle Dokumentart muß natürlich diesen Dokumenttyp kennen. Er erhält aber nur eine Referenz auf ein `Dokument`. Daher ist an dieser Stelle eine gezielte Verletzung des Typsystems nötig: Der Editor muß die ihm übergebene Referenz in eine Referenz vom speziellen Dokument-Typ umwandeln (s. 3.2.2.1).
Ein Standardeditor wird gestartet, wenn es für ein ausgewähltes Dokument keinen speziellen Editor gibt, ist ein allgemeiner Editor vorhanden. Dieser Editor prüft, ob das Dokument diejenigen Methoden anbietet, die zur Bearbeitung mindestens nötig sind (s. 3.2.1.4), konfiguriert sich je nach den vorhandenen Methoden (z.B. als reiner Viewer ohne Änderungsmöglichkeit) und ruft die Dokument-Methoden auf, indem er ihren Namen angibt (s. 3.2.2.2). Hier liegt ebenfalls eine Verletzung des statischen Typsystems vor: Ob das Dokument den richtigen Typ hat, d.h. ob es eine ausreichende Schnittstelle zur Bearbeitung bietet, wird nicht zur Übersetzungszeit, sondern erst zur Laufzeit geprüft.
Damit nicht für Körbe und jede Dokumentart eigene Speicher- und Lademethoden geschrieben werden müssen, wird diese Funktionalität durch generische Speicher- und Laderoutinen realisiert. Das MOP erlaubt den direkten Zugriff auf die Repräsentation der Objekte, also die gezielte Verletzung der Zustandskapselung (s. 3.2.3).
Die Makro-Fähigkeit der Editoren läßt sich generisch realisieren, wenn der Mechanismus zum Methodenaufruf offengelegt ist. Dann wird zur Makroaufzeichnung jeder Methodenaufruf an eine Rekordermethode umgeleitet, die den Aufruf mitprotokolliert, bevor sie die Kontrolle an die gerufene Methode abgibt (s. 3.3.2). Zum Abspielen eines Makros werden die aufgezeichneten Methoden wieder aufgerufen (s. 3.2.2.2).

3.2 Überschreitung der Grenzen streng getypter Sprachen

Die Verwendung einer streng getypten Sprache hat insbesondere zwei Vorteile:

- Viele Programmierfehler (Inkonsistenzen innerhalb des Quelltexts) werden schon vom Compiler als Typfehler identifiziert und können sich daher zur Laufzeit nicht mehr auswirken.
- Durch die Typisierung werden Informationsbarrieren erzeugt, die den Entwickler dazu zwingen, sich auf die vorgegebenen Programmierschnittstellen zu beschränken. Ein Beispiel dafür ist die Modellierung des Zustands eines Objekts. Der Entwickler kann zwar die zur Repräsentierung deklarierten Exemplarvariablen im Quelltext sehen, er darf aber nicht auf sie zugreifen. Hier sieht man deutlich, daß das Verbergen von Information eine erwünschte Eigenschaft streng getypter Sprachen ist, denn es soll ja gerade sichergestellt werden, daß der Zustand nur mittels der definierten Schnittstelle manipuliert wird.

Es gibt allerdings einige Ausnahmefälle, in denen sowohl die Typkontrollen als auch die Informationsbarierren zu unerwünschten Hindernissen werden. Wenn eine Sprache für diese Ausnahmefälle keine Vorsorge trifft, entsteht ein paradoxer Effekt: Der Entwickler wird gezwungen, im ersten Fall das Typsystem nicht voll auszunutzen, im zweiten Fall die vor ihm verborgene Information selbst noch einmal in das Programm hineinzucodieren. Die Verantwortung für die Konsistenz, die dem Entwickler eigentlich durch die Typisierung soweit wie möglich abgenommen werden sollte, muß er nun wie im Fall einer schwach getypten Sprache wieder selbst tragen.

Hier schafft ein Metaobjekt-Protokoll Abhilfe. Es bietet im ersten Fall die Möglichkeit, das Typsystem gezielt zu verletzen, im zweiten Fall öffnet es quasi eine Hintertür zu der benötigten Information. Zwar ist auch hier die Gefahr des Auftretens von Fehlern gegeben, sie wird aber durch zweierlei gegenüber der Lösung „Marke Eigenbau“ vermindert:

- Erstens ist die vom MOP zur Verfügung gestellte Information immer fehlerfrei, da sie automatisch aus dem aktuellen Quelltext extrahiert wird[1].
- Zweitens kann ein MOP in den Fällen, in denen das statische Typsystem umgangen wird, Laufzeitprüfungen erzwingen und dadurch einen etwaigen Fehler zum frühestmöglichen Zeitpunkt aufdecken.

Außerdem spart sich der Entwickler die Mühe, Informationen ein zweites Mal zu modellieren, die eigentlich schon im System vorhanden ist.

3.2.1 Aufheben von Informationsbarrieren

Der erste Schritt in einem Metaobjekt-Protokoll ist, die verborgene Information über eine wohldefinierte Abfrage-Schnittstelle wieder zur Verfügung zu stellen.
Diese Abfragen lassen sich in vier Bereiche unterteilen:

- Abfragen über die Klassenzugehörigkeit (Runtime Type Information)
- Abfragen über Vererbungsbeziehungen
- Abfragen über die Repräsentation des Objektzustandes
- Abfragen über die Schnittstelle

In allen vier Fällen werden die Abfragen nur als Hilfsmittel zur Realisierung anderer Funktionalität verwendet.

3.2.1.1 Runtime Type Information (RTTI)

Die Ausnutzung der Polymorphie bringt es mit sich, daß die Referenz, die man auf ein Objekt hält, nicht denselben Typ hat wie das Objekt selbst, sondern einen Supertyp des Objekttyps. Um die ursprüngliche Typinformation wiederzugewinnen, bietet daher jedes MOP die Möglichkeit, die Klasse eines Objekts zu erfragen. Die entsprechende Methode bzw. Funktion liefert das Klassen-Metaobjekt der Klasse des Objekts zurück. Ein Klassen-Metaobjekt hat mindestens

- eine eindeutige Id;
- eine Repräsentation des Klassennamens als Zeichenkette.

Der Zugriff auf diese beiden Informationen ermöglicht den Klassenvergleich zweier Objekte und das Testen der Klasse eines Objekts, bevor ein Cast auf eine Unterklasse versucht wird.
Meistens bietet das Klassen-Metaobjekt auch den Zugang zu weiteren Meta-Informationen über die Klasse.

3.2.1.2 Abfragen über Vererbungsbeziehungen

Information über Vererbungsbeziehungen wird üblicherweise zugänglich gemacht, indem die Klassen-Metaobjekte so verbunden werden, daß sie zur Laufzeit einen dem Klassenbaum entsprechenden Objektbaum darstellen. Dieser Baum kann dann mit den üblichen Baum-Operationen traversiert und so die gewünschte Information extrahiert werden. Für bestimmte häufig gebrauchte Abfragen sind meist ensprechende Methoden schon vorhanden. Solche Abfragen sind u.a.:

- Ist die Klasse D von C abgeleitet?
- Ist die Klasse D eine Oberklasse von C?
- Von welchen Klassen erbt die Klasse C direkt?
- Welche Klassen sind von C abgeleitet?

Falls es, wie in C++, verschiedene Arten der Vererbung gibt, müssen die Kanten des Objektbaums entsprechend attribuiert werden.
Information über Vererbungsbeziehungen wird üblicherweise nie isoliert benötigt, sondern immer im Zusammenhang mit anderer Information. Um z.B. den Zustand eines Objekts komplett abspeichern zu können, müssen auch die Exemplarvariablen der Superklassen abgespeichert

[1] Dabei wird natürlich vorausgesetzt, daß der MOP-Generator fehlerfrei ist.

werden. Dafür benötigt man die Information, welche Superklassen eine Klasse hat. Für Anweisungen wie „führe Methode X für alle Superklassen von C aus“ gibt es oft spezielle Methoden, die die Vererbungsinformation als implizite Information zur Verfügung stellen, wie z.B. `allSuperclassesDo` in Smalltalk

3.2.1.3 Abfragen über die Repräsentation des Objektzustandes

Auf Informationen über die Repräsentation hat der Entwickler über eine Methode Zugriff, die ihm ein sog. Field Dictionary liefert. Dies ist eine Liste oder ein Array von Variablen-Metaobjekten, die (je nach Sprache unterschiedlich) für jede Exemplarvariable folgende Informationen enthalten können:

- Name der Variablen;
- Typ der Variablen;
- Größe der Variablen;
- sonstige Attribute.

Je nach Sprache sind erheblich weniger Informationen nötig; in Sprachen, in denen es nur Objekte gibt, wie z.B. Smalltalk, benötigt man nur den Namen, da alle Variablen Objekte sind und es keine Typen gibt.
Die Information über die Repräsentation wird zur Realisierung des Object-I/O benötigt (s. 3.2.3).

3.2.1.4 Abfragen über die Schnittstelle

Analog zum Field Dictionary gibt es oft auch ein Method Dictionary. In diesem Dictionary sind die Informationen über jede Methode, die die Klasse hat, verzeichnet:

- Name der Methode;
- Signatur der Methode;
- sonstige Attribute.

Diese Information wird im Zusammenhang mit dem dynamischen Methodenaufruf verwendet.

3.2.2 Verletzung des Typsystems

Die oben genannten Abfragen können nur im Zusammenhang mit einer gezielten Verletzung des Typsystems oder der Zustandskapselung sinnvoll eingesetzt werden. Dieser Abschnitt beschäftigt sich mit Verletzungen des Typsystems, der nächste mit Verletzungen der Zustandskapselung.
Ein MOP kann zwei Arten der Verletzung des Typsystems erlauben:

- Wiedergewinnung des ursprünglichen Typs eines Objekts;
- Zugriff auf die Schnittstelle eines Objekts unter Umgehung der statischen Typprüfung.

3.2.2.1 Wiedergewinnung der ursprünglichen Typinformation (Downcasting)

Die Information über den eigentlichen Typ eines Objekts (RTTI) kann in getypten Sprachen nicht nutzbar gemacht werden, wenn keine Typumwandlung erlaubt ist. Um die volle Schnittstelle des Objekts nutzen zu können, ist die gezielte Verletzung des Typsystems durch Umwandeln einer Objekt-Referenz auf eine Referenz eines Subtyps nötig. Daher bieten die MOPs getypter Sprachen für diese Umwandlung einen speziellen Downcast-Operator. Dieser Operator greift intern auf die RTTI zurück und prüft die Zulässigkeit des Casts, bevor er ihn durchführt. Den Downcast-Operator benötigt man in folgenden Situationen:

- In einem spezialisierten Team von Klassen erhält ein Partner die Referenz auf einen anderen Partner als Referenz mit dem Typ der abstrakten Basisklasse.
- Ein Behälter, der Objekte verschiedener Klassen enthält, kann nur eine Referenz auf die gemeinsame Basisklasse der enthaltenen Objekte zurückgeben. Wenn auf ein aus dem Behälter genommenes Objekt über seine volle Schnittstelle zugegriffen werden muß, ist ein Downcast erforderlich.
- Ein Spezialfall ist die Einrichtung einer systemweiten Objekttabelle, die Referenzen auf alle im System vorhandenen Objekte enthält und ständig aktualisiert wird ([Gamma92]). Mit Hilfe einer solchen Tabelle kann man Datenbank-artige Abfragen über die Objekte realisieren (z.B. „gib mir alle Objekte der Klasse Person, deren Name mit einem ‘E’ beginnt“). Dazu muß die Klassenzughörigkeit jedes Objekts abfragbar sein und die Referenzen der

Objekte, die typkompatibel zur abgefragten Klasse sind, müssen auf den abgefragten Typ gecastet werden (im Beispiel von einer Referenz auf Object zu einer Referenz auf Person)

3.2.2.2 Aufheben der statischen Typprüfung (Dynamischer Methodenaufruf)

Noch größere Flexibilität gewinnt man, wenn erst zur Laufzeit geprüft wird, ob eine bestimmte Methode, die aufgerufen werden soll, überhaupt existiert und ob die Typen der übergebenen Argumente zu den Typen der formalen Parameter der Methode passen (Dies ist in schwach getypten Sprachen wie CLOS und Smalltalk der Standard). Die Verletzung des Typsystems geschieht an zwei Stellen:

- Der Typ des Objekts, dessen Methode aufgerufen wird, wird nicht statisch geprüft.
- Der Typ der Argumente wird nicht statisch geprüft.

Um diese Flexibilität zu ermöglichen, muß ein MOP erstens ein Methoden-Dictionary zur Verfügung stellen, das Abfragen über das Vorhandensein von Methoden und ihre Signatur erlaubt. Zweitens muß die derart flexibilisierte Klasse eine Methode haben, die als Argumente einen Methodennamen und weitere Parameter erhält; sie ruft die Methode, die den übergebenen Namen trägt, mit den übergebenen Parametern auf.
Eine derartige Flexibilität ist in folgenden Fällen nötig:

- Ein bestimmter „Typ“ ist nicht mit Hilfe einer Klasse modelliert, von der alle Klassen erben, die diesen Typ haben, sondern durch eine Schnittstelle, die alle „typkonformen“ Objekte bieten müssen.
- Methodenaufrufe sollen reifiziert werden, z.B. um sie zu wiederholen (Makros) oder an andere Rechner weiterzuleiten (Verteilte Objektsysteme).
- Eine Anwendung soll nach „außen“ geöffnet werden, d.h. es soll ermöglicht werden, daß andere Anwendungen beliebige öffenliche Methoden aufrufen können. Z.B. kann ein externer Interpreter zur Programmierung häufig vorkommender Abläufe (Scripting) verwendet werden [KBSW94].

3.2.3 Verletzung der Zustandskapselung

Üblicherweise ist der direkte Zugriff auf den Zustand von Objekten unter Umgehung ihrer Schnittstelle eine der größten softwaretechnischen Sünden, die man begehen kann. Es gibt jedoch einige Fälle, in denen gerade dieser direkte Zugriff softwaretechnisch sinnvoll ist:

- Generische Vergleichs- und Kopieroperationen;
- Objekt-Ein-/und Ausgabe, z.B. zum Speichern in Dateien, Anschluß an relationale Datenbanken, Transportieren von Objekten in einem verteilten System, usw.;
- Garbage Collection.

In diesen Fällen läßt sich, falls der Zugriff auf die Repräsentation möglich ist, ein für alle Klassen gültiger Algorithmus angeben. Das Verbot des Zugriffs auf den Zustand führt dagegen dazu, daß für jede Klasse die entsprechenden Methoden neu programmiert werden müssen.
Der Zugriff auf den Zustand kann auf zwei Arten erfolgen:
Die erste Variante besteht darin, die benötigte Information implizit zur Verfügung zu stellen, indem die nötigen Methoden automatisch generiert werden. Dieser Ansatz hat mehrere Vorteile: Erstens muß der Entwickler nichts selbst programmieren. Zweitens kann die Kapselung der Objekte durch den Verwender des MOP nicht mehr verletzt werden. Drittens wird insbesondere bei Compiler-Sprachen die Performance deutlich besser. Der Nachteil besteht darin, daß die Zugriffsoperationen vorgegeben sind und vom Entwickler nicht mehr modifiziert werden können.
Die zweite Variante besteht darin, dem Entwickler die Repräsentation als Exemplarvariablenbehälter zur Verfügung zu stellen (explizite Meta-Information). Dann kann er selbst Methoden schreiben, die über die Variablen-Metaobjektliste iterieren und für jede Variable eine bestimmte Aktion durchführen (Kopieren, Vergleichen, in DB speichern, usw.). Nachteile der expliziten Variante sind die nicht mehr durchführbare Typprüfung und ein verschlechtertes Laufzeitverhalten.
Da es keine eindeutige Entscheidung zugunsten der einen oder anderen Variante gibt, enthält ein MOP oft beide Ansätze: In Eiffel z.B. werden einerseits Vergleichs-, Kopier- und Ein-/Ausgabeoperationen für jede Klasse ohne Zutun des Entwicklers zur Verfügung gestellt, andererseits hat

der Entwickler aber über die Klasse INTERNAL auch Zugriff auf die explizite Information (s. 5.1).

3.3 Aufbrechen der „Black Box“-Sicht der Sprachimplementierung

Wenn auch der Softwareentwickler üblicherweise froh ist, daß er über die Implementierung einer Sprache nichts wissen muß, so gibt es doch manche Situationen, in denen er einen Eingriff in die Implementierung vornehmen möchte. Dazu kann ihm ein MOP eine wohldefinierte Schnittstelle zur Verfügung stellen, die eine Umlenkung des Kontrollflusses innerhalb des Laufzeitsystems gestattet. Dies ist an zwei Stellen interessant:

- Speicherung der Exemplarvariablen;
- Methodenaufruf.

In beiden Fällen verschlechtert sich das Laufzeitverhalten eines übersetzten Programms stark, sobald solch eine Kontrollflußumlenkung vorgenommen wird. Daher bieten nur die MOPs interpretativer Sprachen solch eine Schnittstelle. Die einzige Ausnahme ist Open C++, ein MOP, das zur Unterstützung verteilter Objekte dient; in diesem Fall ist die Übertragungszeit von einem Rechner zum anderen der dominierende Faktor, sodaß die Verschlechterung des Laufzeitverhaltens durch das MOP vernachlässigbar ist (s. 6.8).

3.3.1 Eingriff in die Speicherung von Exemplarvariablen

Um den Speicherungsalgorithmus offenzulegen, bedarf es nur zweier zusätzlicher Methoden: Einer Methode zum Lesen einer Exemplarvariablen und einer weiteren zum Setzen. In Smalltalk z.B. gibt es die Methode `instVarAt: i`, die den Wert der `i`-ten Exemplarvariablen zurückliefert. Die Methode `instVarAt: i put: value` weist der `i`-ten Exemplarvariablen den Wert `value` zu. Diese beiden Methoden werden zu jedem Zugriff auf eine Exemplarvariable aufgerufen. Durch Überladen dieser Methoden läßt sich der Standard-Speicherungsalgorithmus für alle Klassen oder Teile des Klassenbaums durch einen eigenen Algorithmus ersetzen. Dies kann entweder aus Effizienzgründen sinnvoll sein oder wenn Objekte extern gespeichert werden sollen:

- Für Objekte, die viele Exemplarvariablen haben, die jedoch meistens nicht alle benutzt werden, kann es platzsparender sein, die Exemplarvariablen nicht in einem zusammenhängenden Speicherblock abzulegen, sondern für jede Exemplarvariable eine Hash-Table anzulegen und sie dort zu speichern.
- Wenn der Objektzustand unmittelbar bei jeder Änderung in einer Datenbank aktualisiert werden soll, ist ebenfalls ein Eingriff in den Speicherungsmechanismus erforderlich (mit der Information in 3.2.3 läßt sich nur eine verzögerte Speicherung realisieren).

In Abschnitt 5.3.6 wird als Beispiel für eine Anwendung des CLOS-MOP eine automatische Protokollführung über Zugriffe auf Exemplarvariablen vorgestellt.

3.3.2 Eingriff in den Methodenaufruf (Methodenumlenkung)

Der Zugriff auf den Methodenaufruf-Mechanismus wird durch eine Methode ermöglicht, die vor jedem Methodenaufruf die Kontrolle erhält und sie dann wieder abgibt (Eine sog. Around-Methode). In Open C++ ist dies die Methode `Meta_MethodCall()`, die als Argumente den Namen und ein Array der Argumente der zu rufenden Methode erhält. Innerhalb von `Meta_MethodCall()` können vor und nach dem Aufruf der eigentlichen Methode beliebige Aktionen durchgeführt werden. Der Aufruf der eigentlichen Methode geschieht mittels `Meta_HandleMethodCall()`, die das Argument-Array wieder auspackt und die Methode mit diesen Argumente aufruft. Methodenumlenkung ist für folgende Anwendungsbereiche sinnvoll:

- Verteilte Objekte. In diesem Fall bietet der Mechanismus die Möglichkeit, Methodenaufrufe an ein Objekt auf einem anderen Rechner umzulenken.
- Protokollieren von Methodenaufrufen. Dies ist für Makro-Aufzeichnungen nutzbar, kann aber auch beim Optimieren und Debuggen von Programmen hilfreich sein.

In CLOS ist diese Eingriffsmöglichkeit besonders ausgefeilt: Erstens lassen sich beliebig viele Around-Methoden ineinander verschachteln; zweitens gibt es außer den Around-Methoden als

„syntaktischen Zucker“ auch noch Before- und After- Methoden, die jeweils vor oder nach jedem Methodenaufruf ausgeführt werden.

3.4 Ermöglichen von Reflexion

Nach [Maes87] hat eine Programmiersprache eine reflexive Architektur, wenn folgende Bedingungen erfüllt sind:

- Der Interpreter der Sprache muß jedem laufenden Programm den Zugriff auf die Datenstrukturen ermöglichen, die das Programm repräsentieren.
- Der Interpreter muß garantieren, daß die kausale Beziehung zwischen diesen Datenstrukturen und den Eigenschaften des Programms immer erhalten bleibt, d.h. eine Veränderung der Datenstrukturen muß zu einer entsprechenden Veränderung des laufenden Programms führen.

Diese beiden Eigenschaften werden üblicherweise realisiert, indem der Interpreter zum Bestandteil des MOP gemacht wird. Ein benutzbares Modell des Interpreters ermöglicht selbstmodifizierende Programme (Smalltalk). Wenn für das Interpreter-Modell ebenfalls die obigen Reflexivitätsbedingungen gelten, kann man durch Modifikation der Interpreter-Modells sogar die Sprache ändern (CLOS).

3.4.1 Ermöglichen selbstmodifizierender Programme

Ein MOP, das eine Schnittstelle zur Veränderung des Programm-Modells bietet, ermöglicht selbstmodifizierende Programme. In diesem Fall gehört oft die Entwicklungsumgebung ebenfalls zum Laufzeitsystem; der Entwickler deklariert also z.B. neue Klassen, indem er eine Erzeugungs-Methode der Klassen-Metaklasse aufruft.
Auch für das Modifizieren von Klassen gibt es entsprechende Methoden. Sowohl in Smalltalk als auch in CLOS lassen sich die Vererbungsbeziehungen im nachhinein verändern. So kann man z.B. in Smalltalk einem Metaklassenobjekt `C` die Nachricht `superclass: D` schicken; das hat zur Folge, daß von nun an die Klasse `C` von `D` erbt. Solche Änderungen sind natürlich nur in nicht streng getypten Sprachen erlaubt, da eine Prüfung auf Typverletzungen durch die Modifikation der Vererbungsbeziehung nicht mehr möglich ist.
Das Hinzufügen und Löschen von Exemplarvariablen und Methoden zur Laufzeit ist natürlich ebenfalls möglich.
Diese Programmodifikationsschnittstelle kann auch vom Programm selbst aufgerufen werden. Dafür gibt es u.a. folgende Anwendungsfälle:

- Es können automatisch bestimmte Methoden, z.B. für Object-I/O (s.o.) zur Klassendeklarationen hinzugefügt werden.
- Es lassen sich selbst-optimierende Programme realisieren
- Lernende Systeme können ihre Algorithmen verbessern

Die extensive Nutzung der Selbstmodifikation hat zur Folge, daß sich solche Programme kaum noch durch das Lesen des Programmcodes verstehen lassen (besonders, wenn er sich laufend ändert).

3.4.2 Ermöglichen der Modifikation der Sprache

Noch weitergehende Möglichkeiten werden dem Entwickler geboten, wenn das System nicht nur ein Modell des Programms enthält, sondern auch ein Modell des Interpreters der Sprache, das die beiden obigen Bedingungen für reflektive Architekturen erfüllt. Dann kann durch Änderungen dieses Modells die Sprache geändert werden.
Da der Interpreter sich quasi selbst interpretieren muß, entsteht eine rekursive Struktur. Daher heißt ein solcher Interpreter „metazirkulärer Interpreter“. Um nicht in eine rekursive Endlosschleife zu geraten, gibt es einen (in einer anderen Sprache geschriebenen) Basis-Interpreter, der das Standardverhalten implementiert. Bei jedem Schritt der Ausführung wird geprüft, ob eine für diesen Schritt relevante Änderung des Interpreter-Modells durchgeführt worden ist. Wenn dies der Fall ist, wird das Interpreter-Modell für die weitere Ausführung verwendet; wurde nichts geändert, so kann der Basis-Interpreter die fest einprogrammierte Standardvariante ausführen.

Beim Ändern des Interpreter-Modells muß der Anwender darauf achten, daß sich alle Operationen auf das Standardverhalten zurückführen lassen.
Meiner Meinung nach ist die einzige sinnvolle Anwendung für diese Funktionalität, Sprachvarianten ausprobieren zu können, ohne großen Aufwand für die Implementierung des Interpreters investieren zu müssen. Eine solche prototypische Vorgehensweise ist für die Sprachentwicklung sehr angebracht, da sich die Nützlichkeit und mehr noch die Gefährlichkeit von Sprachfeatures erst in der Verwendung der Sprache zeigt.
Für jedes Anwendungsprojekt einen eigenen Sprachdialekt zu erzeugen, scheint dagegen problematisch, weil dann zum Verstehen des Programms nicht nur die Analyse des Quelltexts nötig ist, sondern zusätzlich noch das Erlernen des neuen Dialekts.

4 Meta-Konzepte – Entwurfsdimensionen

Beim Entwurf eines Metaobjekt-Protokolls gibt es Spielräume, auf welche Art man die Schnittstelle gestaltet. Im folgenden sollen diese Spielräume und die Vor- und Nachteile der einzelnen Entwurfsentscheidungen aufgezeigt werden.

4.1 Explizite vs. implizite Meta-Information

Meta-Information kann sowohl explizit als auch implizit zur Verfügung gestellt werden. Dies läßt sich sehr schön an einem Beispiel aus C/C++ verdeutlichen, nämlich der dynamischen Allozierung von Objekten. In C gibt es über die Größe des benötigten Speicherplatzes explizite Meta-Information, auf die mit `sizeof()` zugegriffen werden kann. Speicherplatz für ein Objekt alloziert man folgendermaßen:

```
struct Point{  /* C kennt keine Klassen */
    int x;
    int y;
};

Point* p;
p = malloc( sizeof( Point ) );
```

Dasselbe läßt sich in C++ mit `new` erreichen:

```
p = new Point;
```

In diesem Fall steht die Meta-Information über den zur Repräsentation des Objekts benötigten Speicherplatz nur implizit zur Verfügung. Sie ist quasi im Operator `new` gekapselt.
Der Vorteil der expliziten Variante liegt in der Flexibilität. So läßt sich eine flache Kopie eines Objekts ebenfalls mit Hilfe von `sizeof()` erzeugen:

```
Sample* p2 = malloc( sizeof( Sample ) );
memcpy( p2, p, sizeof( Sample ) );        //bitweise kopieren
```

Der Vorteil der impliziten Lösung liegt dagegen in der Einfachheit und in der größeren Sicherheit durch die Kapselung (Größe garantiert korrekt, Typprüfung bei der Zuweisung möglich). Außerdem ist Funktionalität, die auf expliziter Information arbeitet, prinzipbedingt erheblich langsamer als implizite Information durch Code-Erzeugung zur Übersetzungszeit, da die Extraktion der Information bei jedem Aufruf wieder durchgeführt werden muß (manchmal helfen Cache-Implementationen). Sofern möglich, ist daher die Realisierung als implizite Information vorzuziehen.

4.2 Statische vs. dynamische Meta-Information

Statische Meta-Information wird vor oder zu Beginn der Laufzeit eines Programms erzeugt und ändert sich während des Programmlaufs nicht mehr. Dazu gehört in statisch getypten Sprachen z.B. Klasseninformation (Superklassen, Größe von Exemplaren, usw.).
Dynamische Meta-Information bietet den Zugriff auf Programmzustände, die sich während der Laufzeit ständig ändern, und wird daher immer wieder aktualisiert (z.B. Exemplar-Behälter, die alle Exemplare einer Klasse enthalten).
In interpretativen Sprachen wie CLOS, in denen zur Laufzeit sogar dann eine Klassendefinition dynamisch geändert werden kann, wenn bereits Exemplare dieser Klasse erzeugt worden sind, gibt es prinzipiell nur dynamische Meta-Information.

4.3 Information zur Übersetzungszeit vs. Information zur Laufzeit

Ein erheblicher Unterschied besteht auch darin, ob die Meta-Information dem Programmierer schon in seinem Quelltext zur Verfügung steht, also zur Übersetzungszeit, oder ob sie erst während der Laufzeit eines Programms abgefragt werden kann. Meta-Informationen zur Über-

setzungszeit dienen zur Code-Generierung, während Meta-Information zur Laufzeit Zugriff auf den dynamischen Zustand des Programms bieten.
Ein typisches Beispiel für Meta-Information zur Übersetzungszeit ist in C++ der `sizeof()`-Operator. Die damit abgefragte Größe von Objekten läßt sich z.B. in Variablendeklarationen verwenden (Beispiel aus [WP95]):

```
// T is a class
#define N sizeof(T) // fixed at compile time
char buf[N];
T obj; // obj initialized to its original value
memcpy(buf, &obj, N);
...
```

Eine Voraussetzung dafür, Meta-Information schon zur Übersetzungszeit zur Verfügung zu stellen, ist, daß sie während der gesamten Laufzeit des Programms konstant bleibt. Meta-Information schon zur Übersetzungszeit zur Verfügung zu stellen, hat folgende Vorteile:

- Der generierte Code durchläuft die Typprüfung des Compilers.
- Die Verwendung der sichereren und effizienteren impliziten Meta-Information wird gefördert.

Bisherige MOPs für C++ liefern erst zur Laufzeit Information, während MopGen schon zur Übersetzungszeit Informationen liefert.

4.4 Der Umfang: Inspektion - Extension - Modifikation

Es gibt drei Stufen der Funktionalität, die ein MOP bieten kann: Inspektion, Extension und Modifikation. Sie werden im folgenden beschrieben.

4.4.1 Inspektion

Die niedrigste Stufe eines MOP ist die Inspektion. Auf dieser Stufe werden Informationen über das Programm je nach Sprache als Klassen, Objekte oder Funktionen zur Verfügung gestellt. Typisches Beispiel sind die Universal Features in Eiffel, die es mit Hilfe der vom Compiler generierten Methode `conforms_to()` ermöglichen, den dynamischen Typ eines Objekts herauszubekommen (s. 5.1). Die nur abfragenden Methoden der Klassenobjekte in Smalltalk gehören ebenfalls zum Inspektionsteil des MOP (s. 5.2).

4.4.2 Extension

Die nächste Stufe ist die Extension. Hier bietet das MOP die Möglichkeit, zusätzliche Programmteile zu generieren, wie z.B. Standardmethoden oder neue Metaobjekte. Die vom Entwickler geschriebenen Programmteile können nicht per MOP geändert werden, sie können aber auf die mit dem MOP generierten Methoden oder Objekte zugreifen. Zu dieser Kategorie gehören z.B. die mit Hilfe des C-Preprozessors entwickelten MOP's der großen C++-Frameworks (s. 6.5).

4.4.3 Modifikation

Wenn zur Laufzeit auf den Interpreter einer Sprache zugegriffen werden kann, wird die Modifikation des Programms zur Laufzeit möglich. In diesem Fall können zur Laufzeit z.B. neue Methoden zu vorhandenen Klassen hinzugefügt, neue Klassen erzeugt oder die Vererbungshierarchie geändert werden. Diese Möglichkeiten werden u.a. benötigt, wenn Entwicklungsumgebung und Anwendungsprogramm eine Einheit bilden, wie z.B. bei Smalltalk.

4.5 Realisierung mittels eines metazirkulären Interpreters

In einer OO-Sprache mit metazirkulärem Interpreter sind die Metaobjekte nicht nur Informationsträger, die ein Abbild des eigentlichen Programms geben, sondern sie sind das Programm. Anders gesagt: der metazirkuläre Interpreter interpretiert die Metaobjekte, nicht irgendwelche internen Strukturen. Das hat zur Folge, daß eine Änderung dieser Objekte eine Programmänderung ist. Da der Interpreter selbst ebenfalls Bestandteil des Laufzeitsystems ist, ist

es auch möglich, den Interpreter und damit die Sprache zu modifizieren[1]. Beispiele für OO-Sprachen mit metazirkulärem Interpreter sind CLOS (sofern man CLOS als objektorientierte Sprache ansieht) und 3-KRS [Maes87]. Die Realisierung des MOP mit Hilfe eines metazirkulären Interpreters ermöglicht die größte Funktionalität, hat jedoch auch Nachteile:

- Es ist keine reine Compilersprache möglich.
- Beim heutigen Stand der Technik ist keine Typprüfung realisierbar.
- Die Schnittstelle der Metaklassen muß einerseits das Programm so modellieren, daß es für den Softwareentwickler gut verständlich wird, andererseits muß sie aber auch eine effiziente Ausführung des Programms erlauben. Diese beiden Eigenschaften lassen sich nur schwer unter einen Hut bringen.

4.6 Gestaltung der Schnittstelle

Die Schnittstelle eines MOP läßt sich auf dreierlei Art und Weise gestalten:

- als spezielle Anweisungen der Programmiersprache (z.B. Downcast-Operator in Eiffel, s. 5.1.3);
- als Methoden einer Standard-Basisklasse, von der alle Anwendungsklassen erben;
- als Methoden eigener Metaklassen.

Die erste Variante ist in folgenden Fällen angebracht:

- Der Downcast-Operator sollte eine Sprachanweisung sein, da eine entsprechende Methode nur eine Klassenmethode sein kann und für jede Klasse generiert werden müßte.
- Falls der Entwickler die Freiheit behalten soll, nicht alle Klassen von einer Basisklasse abzuleiten, sind ebenfalls Sprachanweisungen nötig (z.B. in C++, s. 6.3).

Die zweite Variante ist für RTTI zwingend; manche imlizit realisierte MOP-Funktionalität (z.B. Vergleichs- und Kopieroperationen) sollten auch Bestandteil der Anwendungsklassenschnittstelle sein, da sonst die Benutzung im Vergleich zur dritten Variante unnötig umständlich wäre:

```
// Schnittstelle in der Anwendungsklasse:
if ( Obj1.IsEqual( Obj2 ) ) ...

// Schnittstelle in der Metaklasse:
if ( Obj1.GetMeta().TestEquality( Obj1, Obj2 ) ) ...
```

Explizite Meta-Information sollte dagegen auf jeden Fall über Metaklassen zur Verfügung gestellt werden. Nur so kann sie konsequent objektorientiert modelliert werden.

[1] Damit es nicht zu einer endlosen Rekursion kommt, ist das Standardverhalten des Interpreters durch einen Basisinterpreter, der in einer anderen Programmiersprache geschrieben ist, realisiert.

5 Meta-Information in Eiffel, Smalltalk und CLOS

Die allgemeinen Betrachtungen in den Kapiteln 3 und 4 werden in diesem Kapitel anhand ausgewählter OO-Sprachen illustriert.
Da es mit möglichst wenig Kenntnis der einzelnen Sprachen verständlich sein soll, aber eine Einfürung in die Sprachen den Rahmen dieser Arbeit sprengen würde, sind die folgenden Informationen z.T. etwas vereinfacht. Die einzelnen Abschnitte sollen nicht als Sprachreferenz dienen, sondern nur die für das MOP der jeweiligen Sprache relevanten Fakten darstellen. Dies gilt insbesondere für die Unterabschnitte mit dem Titel „Das Objektmodell“, die nicht das komplette Modell jeder Sprache beschreiben, sondern nur die hinsichtlich des MOP relevanten Eigenschaften.
Die Sprachen sind einerseits nach ihrem Verbreitungsgrad ausgesucht - dahinter steckt die Vermutung, daß viel verwendete Sprachen auch ein praxistaugliches MOP haben werden -, andererseits sollten möglichst unterschiedliche Sprachen vorgestellt werden, um die Bandbreite der Entwurfsmöglichkeiten für MOPs besser zu veranschaulichen. CLOS durfte nicht fehlen, da das MOP im Zentrum des Entwurfs dieser Sprache stand.

5.1 Eiffel

5.1.1 Das Objektmodell

Eiffel enthält ein relativ kleines und nicht mehr erweiterbares MOP ([Meyer92], [MeyNer90]). Zugriff auf die grundlegende Funktionalität erhält man über die Schnittstelle der Klasse ANY. Von dieser Klasse erbt jede Klasse. Da in Eiffel alle Entitäten Exemplare einer Klasse sind, steht diese Schnittstelle für jedes Objekt zur Verfügung. Um Objekte in Dateien ablegen und lesen zu können, muß eine Klasse von STORABLE erben. Eiffel stellt also fast nur implizite Meta-Information zur Verfügung. Die einzige Ausnahme ist die in der Standardbibliothek enthaltene Klasse INTERNAL, mit deren Hilfe man die Zustandskapselung von Objekten umgehen kann. Metaklassen sind in Eiffel nicht vorhanden.

5.1.2 RTTI

Die Methode `generator: STRING` gibt den Namen der Klasse des Objekts zurück. Mit dieser Information kann man allerdings nichts weiter anfangen (außer sie auszugeben). Da es keine Klassen-Metaobjekte gibt, kann man auch nicht auf sie zugreifen.
Um zu testen, ob ein Objekt typkompatibel zu einem anderen Objekt ist, gibt es die Methode `conforms_to(other: like current): BOOLEAN`.

5.1.3 Downcasting

Downcasting wird in der Eiffel-Terminologi Reverse Assignment genannt. Eine Referenz auf ein Objekt vom Typ ANY kann (versuchsweise) auf eine Referenz einer beliebigen Klasse A zugewiesen werden. Dazu gibt es es den Zuweisungsoperator „?=“. Er führt den Cast durch, falls das referenzierte Objekt Exemplar der Klasse A oder einer abgeleiteten Klasse von A ist, und weist sonst Void zu. Zum Beispiel:

```
x: SAMPLE;
y: NOSAMPLE;
x ?=GetSAMPLEAsANY();  // GetSampleAsAny() returns a reference of type ANY
-- Now x is a reference of an Object of type SAMPLE
y ?=GetSAMPLEAsANY();
-- Now y is Void
```

5.1.4 Compare, Copy, Clone

Um den Entwickler vom direkten Zugriff auf Exemplarvariablen und damit einer Verletzung der Kapselung abzuhalten, erbt jede Klasse von ANY diejenigen Methoden, für die am häufigsten Exemplarvariablenzugriff benötigt wird.

Zum Vergleichen von Objekten gibt es die Methode `equal(some: ANY, other: like some): BOOLEAN`, die Feld für Feld die Gleichheit zweier Objekte prüft, und `deep_equal`, die die komplette Objektstruktur auf Wertgleichheit prüft.
Um zu prüfen, ob zwei Referenzen auf dasselbe Objekt verweisen, muß man die Objekt-Id's vergleichen. Dazu besitzt jedes Objekt eine systemweit eindeutige Id, die im Attribut `object_id` gespeichert ist.
Die Methode `copy(other: like current)` dient dazu, den Wert von `other` auf `current` zu kopieren, während `clone(other: ANY): like other` ein neues Objekt mit dem Wert von `other` zurückliefert. Zu `copy` und `clone` gibt es die Varianten `deep_copy` und `deep_clone`, die das Kopieren und Klonen kompletter Strukturen ermöglichen.

5.1.5 Object-I/O

Für die Realisierung Datei-basierter Persistenz bietet Eiffel die Klasse STORABLE. Exemplare von Klassen, die von STORABLE erben, lassen sich ohne weiteres Zutun des Entwicklers in Dateien ablegen und von dort wieder laden. Dazu gibt es die Methoden `store_by_file( f: FILE )` und `retrieve_by_file( f: FILE )`. `retrieve_by_file()` liest das Objekt wieder ein und weist eine Referenz auf das gelesene Objekt der Exemplarvariablen `retrieved` zu. Durch Zugriff auf `retrieved` erhält man also das zuletzt gelesene Objekt.
Meyer begründet diesen Entwurf damit, daß Funktionen (Routinen mit Rückgabewert) keine Seiteneffekte haben sollten. Das Problem der jetzt gefundenen Lösung liegt darin, daß es bei Parallelverarbeitung unvorhersagbare Resultate liefert.

Damit der Entwickler auch eigene Ein-/Ausgabe-Mechanismen implementieren kann, gibt es die Klasse INTERNAL. Diese Klasse bietet Methoden an, die Informationen über die Exemplarvariablen eines Objekts extrahieren und den direkten Zugriff auf die Exemplarvariablen gestatten. Fast alle Methoden von INTERNAL erwarten als Argumente das Objekt, auf dessen Zustand man zugreifen will, und den Index der Exemplarvariablen, auf die man zugreifen will. Durch Iterieren über die Anzahl der Exemplarvariablen läßt sich der komplette Zustand lesen und schreiben. Die Anzahl der Exemplarvariablen erhält man mit der Methode `field_count( obj: ANY ): INTEGER`..
Zunächst holt man sich mit der Methode `field_type( i: INTEGER; obj: ANY ):INTEGER` den Typ der i-ten Exemplarvariablen. Er ist in einer INTEGER-Zahl codiert. Jeder Klasse ist ein Wert zugeordnet: normale Klassen haben Werte über 0, Built-In-Klassen haben Werte unter 0 (-1: INTEGER, -2: REAL, usw.). Je nach Typ kann man nun eine der folgenden Zugriffmethoden verwenden:

field(i: INTEGER; obj: ANY): ANY	liefert die i-te Exemplarvariable, falls sie ein Exemplar einer normalen Klasse ist
intfield(i: INTEGER; obj: ANY): INTEGER	liefert die i-te Exemplarvariable, falls sie ein INTEGER ist
realfield(i: INTEGER; obj: ANY): REAL	liefert die i-te Exemplarvariable, falls sie ein REAL ist
charfield(i: INTEGER; obj: ANY): CHARACTER	liefert die i-te Exemplarvariable, falls sie ein CHARACTER ist

Für BOOLEANs gibt es spezielle Zugriffsmethoden, da mehrere BOOLEANs in einem Wort gepackt werden. Den Namen einer Exemplarvariablen kann man mit `field_name( i: INTEGER; obj: ANY): STRING` abfragen.

5.1.6 Diskussion

Die beiden Bestandteile des MOP „Reverse Assignment Attempt“ und „INTERNAL“ sind grundlegend:

- Alle anderen beschriebenen Features lassen sich mit ihrer Hilfe in Eiffel formulieren.

- Weder läßt sich der „Reverse Assignment Attempt“-Operator mit Hilfe anderer Sprachkonstrukte ausdrücken noch läßt sich eine Implementation der INTERNAL-Klasse in Eiffel formulieren.

Eine der Designprinzipien von Eiffel ist:

> „It should be possible for a purely static tool to determine the type consistency of every operation by mere examination of the software text, before any execution („static typing“)“ ([Meyer92], S.500).

Die Tatsache, daß die oben erwähnten Bestandteile in Eiffel enthalten sind, obwohl dieses Prinzip dadurch verletzt wird, ist ein Beleg dafür, daß sie für die Entwicklung größerer objektorientierter Systeme unverzichtbar sind. Alle anderen Features dienen im wesentlichen dazu, die Gefährlichkeit der Klasse INTERNAL zu entschärfen: Ihre Implementierung, die Teil der Standardbibliothek ist, kann zwar nicht mehr auf Typverletzungen geprüft werden, aber ihre Benutzung sehr wohl. Für praktisch alle in Eiffel geschriebenen Programme bleibt daher als einziges Typsystem-„Loch“ der „Reverse Assignment Attempt“-Operator.

Es bleiben folgende Kritikpunkte:

- Die Meta-Information ist nicht objektorientiert modelliert, sondern prozedural (Die Klasse INTERNAL ist nichts anderes als eine Sammlung von Routinen: sie hat keinen Zustand, und das zu behandelnde Objekt muß immer als Argument übergeben werden).
- Das Aufheben der statischen Typprüfung beim Methodenaufruf durch dynamischen Methodenaufruf ist nicht möglich[1].
- Die über die Klasse INTERNAL zur Verfügung gestellte Schnittstelle ist für performancekritische Anwendungen (z.B. DB-Anschluß) zu ineffizient. Für solche Anwendungen wäre Meta-Information zur Übersetzungszeit nötig.

[1] Die nicht zu Eiffel gehörende C-Bibliothek Cecil, die es erlaubt, Eiffel-Objekte von C aus zu benutzen, bietet dynamischen Methodenaufruf. Hier eine Beispielfunktion (`eif_rout` ist eine Cecil-Bibliotheks-Funktion):

```
FktPtr eiffel_method;
char*  method_name;
...
eiffel_method = eif_rout( obj, method_name ) // eif_rout liefert fkt-ptr
(*eiffel_method)( obj, arg1, arg2, ...)      // Aufruf der Methode
```

5.2 Smalltalk

5.2.1 Das Objektmodell

Der Aufbau des Objektmodells und des MOP ist durch folgende Regeln gekennzeichnet[1]:

- Jede Klasse ist eine (indirekte) Unterklasse von Object (außer Object selbst).
- Jedes Objekt ist ein Exemplar einer Klasse.
- Jede Klasse ist ein Exemplar einer Metaklasse.
- Jede Metaklasse ist ein Exemplar der Klasse `Metaclass`.

Leider weist die Metaklassen-Konstruktion einige Merkwürdigkeiten auf. Es gibt nicht, wie eigentlich zu erwarten wäre, eine einzige Metaklasse, sondern für jede Klasse eine eigene Metaklasse (Die Klasse '`Sample`' hat als Metaklasse die Klasse '`Sample class`'). Die Klasse ist das einzige Exemplar ihrer Metaklasse. Die Metaklasse wird vom Compiler erzeugt und ist nicht frei wählbar.

Die Vererbungsbeziehung der Metaklassen ist parallel zur Vererbungsbeziehung der Klassen. Alle Metaklassen erben (indirekt) von der Klasse `Class`. Es kommt also zu einer Verdoppelung der Klassenanzahl.
Aus technischen Gründen hat `Class` weitere Oberklassen (`ClassDescription`, `Behavior`), auf die die Metaklassen-Funktionalität verteilt ist. In der Grafik und der Methoden-Tabelle wird dies zur Vereinfachung vernachlässigt.

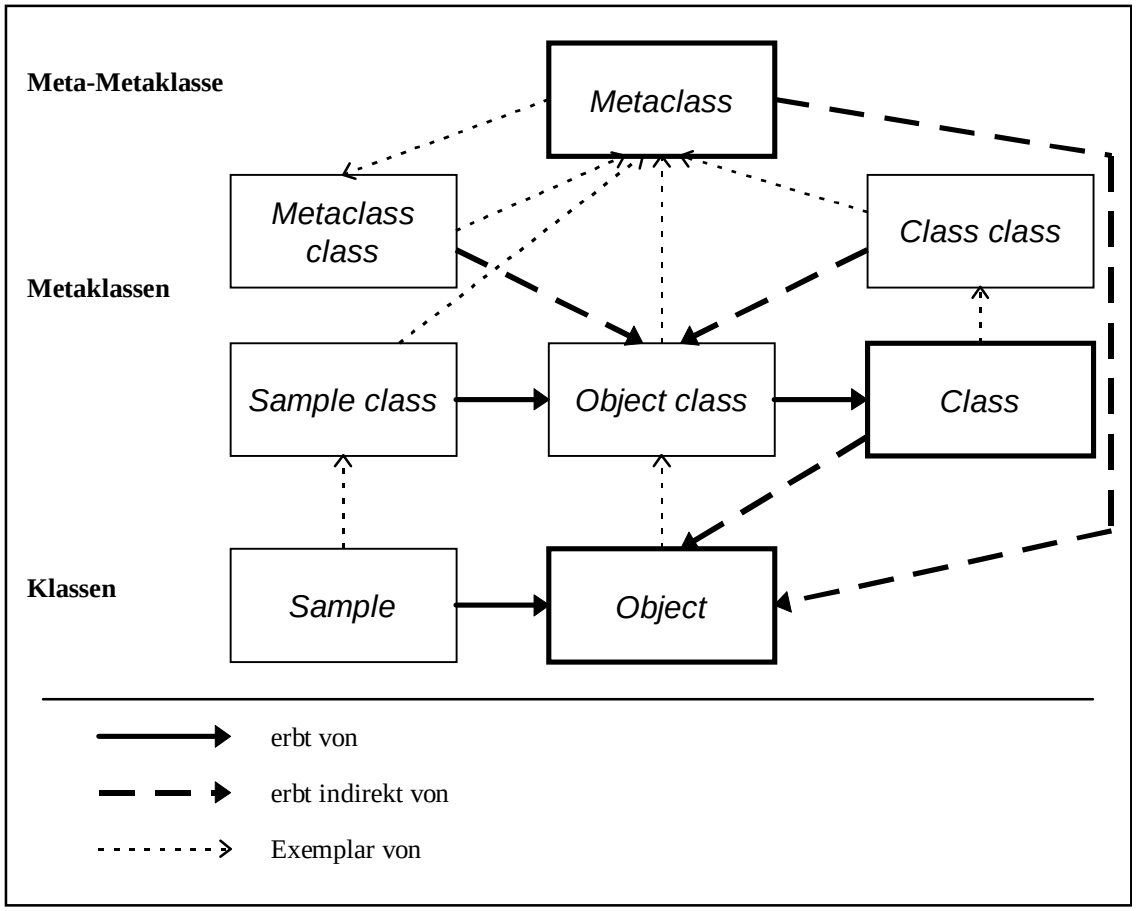

Diagramm 5.1 Klassen- und Objektbeziehungen in Smalltalk

Da zur Laufzeit neue Klassen erzeugt und Methoden zu Klassen hinzugefügt werden können, ist es möglich, mit Hilfe des MOP das Programm zu erweitern und zu modifizieren.

[1] Sämtliche Information über Smalltalk stammt aus [GolRob89].

5.2.2 RTTI

Jedes Objekt kann nach seiner Klasse gefragt werden:

class	liefert das Klassen-Metaobjekt der Klasse des Objekts zurück
isKindOf: aClass	liefert true, wenn das Objekt ein Exemplar der Klasse aClass oder einer von ihr abgeleiteten Klassen ist
isMemberOf: aClass	liefert true, wenn das Objekt ein Exemplar der Klasse aClass ist

Ein Downcasting-Operator ist nicht nötig, da Smalltalk nicht streng getypt ist.

5.2.3 Abfragen

Die Klassen-Metaobjekte haben eine breite Schnittstelle, um Informationen über die Klasse abzufragen und über bestimmte Listen (z.B. die Liste aller abgeleiteten Klassen) zu iterieren[1]:

name	Der Name der Klasse
subclasses	liefert die Menge der direkt abgeleiteten Klassen zurück
allSubclasses	liefert die Menge aller abgeleiteten Klassen zurück
superclass	liefert die direkte Oberklasse zurück
allSuperclasses	liefert die Menge aller Oberklassen zurück
allSubclassesDo: aBlock	führt den Anweisungsblock für alle abgeleiteten Klassen aus
allSuperclassesDo: aBlock	führt den Anweisungsblock für alle Oberklassen aus
instSize	liefert die Anzahl der Exemplarvariablen der Klasse
instVarNames	liefert die Liste der Namen aller Exemplarvariablen, die in dieser Klasse deklariert wurden, zurück
allInstVarNames	liefert die Liste der Namen aller Exemplarvariablen zurück, die in dieser Klasse oder einer ihrer Superklassen deklariert wurden
selectors	liefert die Menge aller Methoden zurück, die in dieser Klasse deklariert wurden
allSelectors	liefert die Menge aller Methoden zurück, die in dieser Klasse oder einer ihrer Superklassen deklariert wurden
allInstances	liefert die Menge aller existierenden Exemplare der Klasse zurück
allInstancesDo: aBlock	führt einen Block für jedes Exemplar der Klasse aus
allSubInstancesDo: aBlock	führt einen Block für jedes Exemplar jeder Subklasse der Klasse aus

5.2.4 Object-I/O

Um die Zustandskapselung zu umgehen, stellt Smalltalk je eine Methode zum Lesen und eine Methode zum Schreiben der Exemplarvariablen zur Verfügung. Diese Methoden nehmen als Parameter nicht den Variablennamen, sondern ihren Index (die an i-ter Stelle deklarierte Variable hat den Index i). Dieses Verfahren wurde wohl aus Effizienzgründen gewählt; auch das Laufzeitsystem liest die Variablen mit diesen Methoden. Mit Hilfe der von `instVarNames` zurückgelieferten Liste läßt sich leicht zu einem Variablennamen der Index finden. Die Methoden sind in der Klasse `Object` deklariert, sodaß alle Klassen sie erben.

instVarAt: i	liefert die i-te Exemplarvariable zurück
instVarAt: i put: value	setzt die i-te Exemplarvariable auf value

[1] Ein Teil dieser Methoden ist schon in den Superklassen von `Class` – `ClassDescription` und `Behavior` – definiert.

Um über alle Exemplarvariablen eines Objekts iterieren zu können, benötigt man noch die Anzahl der Exemplarvariablen; man erhält sie durch Aufruf der Methode `instSize` des Klassen-Metaobjekts.

5.2.5 Dynamischer Methodenaufruf

Der dynamische Methodenaufruf ist mit der Methode `perform` realisiert, die es in mehreren Ausführungen gibt. Sie unterscheiden sich nur durch die Anzahl der mitgelieferten Argumente. Vor dem Aufruf einer Methode mittels `perform` erlaubt `canUnderstand` einen Test, ob die gewünschte Methode überhaupt für das Objekt definiert ist.

canUnderstand: aSymbol	liefert true, wenn das Objekt die Methode aSymbol hat
perform: aSymbol (with: anObject ...)	führt die Methode aSymbol (mit den Argumenten anObject, ...) aus. Es wird geprüft, ob die Anzahl der Argumente mit der Anzahl der formalen Parameter übereinstimmt

`perform` und `canUnderstand` sind Methoden der Klasse `Object`.

5.2.6 Reflexion

Das Laufzeitsystem enthält nicht nur die benötigte Funktionalität zum Ausführen von Programmen, sondern auch alles, was man zum Schreiben und Ändern von Programmen braucht. Die Schnittstelle dafür stellt die Klassenmetaklasse `Class` zur Verfügung:

addInstVarName: aString	fügt eine Exemplarvariable mit dem Namen aString zur Klasse hinzu
removeInstVarName: aStr	entfernt die Exemplarvariable mit dem Na aus der Klasse
addClassVarName: aStrin	fügt eine Klassenvariable mit dem Name Klasse hinzu
removeClassVarName: aSt	entfernt die Klassenvariable mit dem Nam der Klasse
subclass: classNameStri instanceVariableNames stringInstVarNames classVariableNames: stringOfClassVarName poolDictionaries: stringOfPoolNames category: categoryNameString	erzeugt eine neue Subklasse mit dem Namen classNameString, den Exemplarvariablen stringOfClassVarNames, usw.
addSubclass: aClass	fügt die Klasse als abgeleitete Klasse
removeSubclass: aClass	entfernt die Klasse aus der Menge der abgeleiteten Klassen
superclass: aClass	Setzt die Oberklasse auf
addSelector: selector withMethod: compiledM	trägt die schon übersetzte Methode unter dem Selektor (=Namen) in das Meth Dictionary ein
removeSelector: selecto	löscht die Methode mit dem Selektor
compile: code	übersetzt den übergebenen Quelltext trägt d übersetzten Methoden in das Method-Dicti

Mit Hilfe dieser Schnittstelle ist es auch möglich, selbstmodifizierende Program
Z.B. kann eine Datenbankschnittstelle nicht nur mit den in Methoden realisiert
werden, sondern auch, indem für jede Klasse, die persistent gemacht werden soll
Quelltext für Lese-/und Schreiboperationen generiert wird und anschließend di
Methoden mit `compile` und `addSelector` der Klassenschnittstelle hinzugefügt werden.

5.2.7 Diskussion

Durch die Integration des Compilers/Interpreters in das Laufzeitsystem und Öffnen ihrer Schnittstellen für den Entwickler läßt sich mit dem Smalltalk-MOP fast alles machen. Man merkt der Schnittstelle aber an, daß sie eigentlich zur Realisierung des Compilers/Interpreters entworfen wurde und nicht zur Benutzung durch den „einfachen“ Entwickler. Die Aufteilung der Schnittstelle – teils Methoden der Klasse `Object`, teils Methoden der Klassen `Classdescription`, `Behavior`, `Class` und `Metaclass` wirkt vom Benutzungsstandpunkt aus willkürlich, auch wenn es vom Implementationsstandpunkt aus gute Gründe für dieses Design geben mag.
Auch die Entscheidung, für jede Klasse eine eigene Metaklasse einzuführen, ist nur vom Standpunkt des Implementierers aus verständlich.

5.3 CLOS

Das am weitesten ausgebaute MOP bietet CLOS ([KRB91], [Paepcke93]). CLOS ist ein reflexives System, d.h. daß die Metaobjekte, die das Programm beschreiben, zur Implementation gehören; der Interpreter führt das Programm aus, indem er die Information in den Metaobjekten auswertet und ggf. modifiziert. Daher führt eine Änderung der Metaobjekte unmittelbar zu einer Änderung des Programms.

5.3.1 Das Objektmodell

In CLOS sind alle Sprachkonstrukte First-Class-Objekte: Es gibt Klassen-Metaobjekte, „Methoden"-Metaobjekte, Exemplarvariablen-Metaobjekte, usw.
Für das Objektmodell gilt folgendes:

- Jede Entität ist ein Objekt.
- Jedes Objekt ist Exemplar einer Klasse.
- Alle benutzerdefinierten Klasen erben (direkt oder indirekt) von der Klasse `standard-object`[1].
- Jede Klasse ist Exemplar einer Klassen-Metaklasse.
- Alle Klassen-Metaklassen erben von der Klasse `standard-class`.

Standardmäßig ist eine Klasse ein Exemplar der Klasse `standard-class` (die Klasse `sample2` im Diagramm 5.2 ist ein Beispiel dafür). Der Entwickler kann jedoch eigene Metaklassen von `standard-class` ableiten. Ein Beispiel wäre eine Metaklasse, die Persistenz realisiert (`persistent-class`), indem sie die Zugriffsmethoden für Exemplarvariablen überlädt. Klassen erhalten die Persistenz-Eigenschaft dadurch, daß sie als Exemplare dieser neuen Metaklasse erzeugt werden. Die Vererbungsbeziehungen der Klassen werden dadurch nicht beeinflußt (die persistente Klasse `sample1` erbt wie `sample2` von `standard-object`).

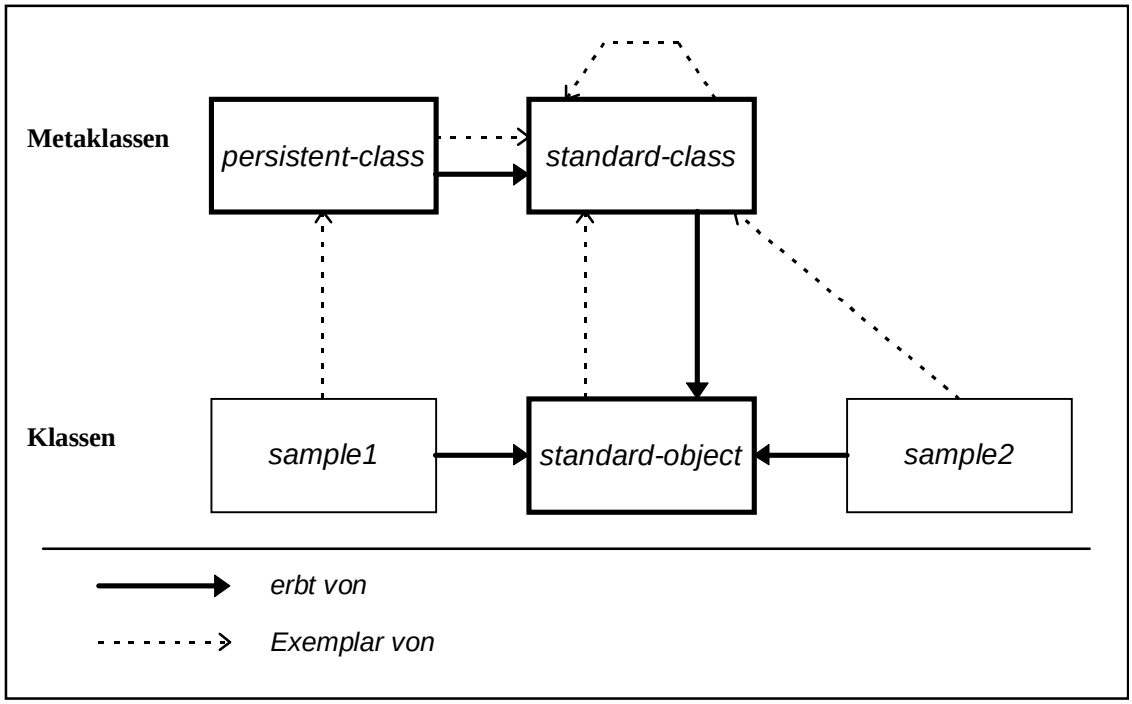

Diagramm 5.2 Klassen und Metaklassen in CLOS

Eine Besonderheit des Objektmodells von CLOS ist, daß die Methoden nicht einzelnen Klassen zugeordnet sind, sondern sog. generischen Funktionen. Eine generische Funktion ist durch den

[1] Die Wurzel des Klassenbaums in CLOS ist eigentlich die Klasse `t`. Von dieser Klasse sind alle Built-In-Klassen - die Datentypen von Common LISP - abgeleitet, wie `symbol`, `list`, `number`, `character`, usw. `standard-object` ist ebenfalls direkt von `t` abgeleitet.

Funktionsnamen gekennzeichnet. Dieser Ansatz hat den Vorteil, daß der polymorphe Methodenaufruf nicht nur über das erste Argument (in anderen OO-Sprachen die implizite this/self-Referenz), sondern über beliebig viele Argumente gesteuert werden kann.
Jede generische Funktion ist durch ein eigenes Funktionsobjekt repräsentiert. Wenn eine Methode mit einer neuen Signatur definiert wird, wird sie in die Methodenliste ihres Generic-Function-Metaobjekts eingetragen. Welche Methode(n) aufgerufen wird(werden), entscheidet sich zur Laufzeit anhand der übergebenen Argumente: Das zuständige Generic-Function-Metaobjekt ruft diejenige Methode auf, deren Argumenttypen am besten zu den aktuell übergebenen Argumenten passen. Das MOP bietet die Möglichkeit, selbst festzulegen, wie das Kriterium „paßt am besten“ berechnet werden soll.
Die Klassendefinitionen enthalten also nur die Festlegung der Vererbungshierarchie (von Subtyp-Hierarchie kann man nicht sprechen, da CLOS bestenfalls schwach getypt ist) und die Festlegung der (logischen) Zustandsrepräsentation. Dazu können Exemplarvariablen deklariert werden, die in CLOS „Slots“ heißen. Die Slots der Superklassen werden im allgemeinen mitvererbt. CLOS hat allerdings insofern einen Fehler in der Sprachdefinition, als bei einer Namenskollision eines neu deklarierten Slots mit dem einer Superklasse der Slot der Superklasse überschrieben wird. Dadurch geht die Abstraktion von der Superklassen-Implementierung verloren[1].
Das MOP von CLOS ist so umfangreich, daß es den Raum sprengen würde, es hier komplett zu beschreiben. Daher werde ich nur kurz die Realisierung der üblichen Funktionalität darstellen und anschließend ein Beispiel geben, das zeigt, wie einfach der Anwendungsentwickler das MOP nutzen kann.

5.3.2 RTTI

Für jede Klasse gibt es ein Metaobjekt; dieses Klassen-Metaobjekt ist standardmäßig ein Exemplar der Metaklasse `standard-class`, der Entwickler kann jedoch auch eigene Metaklassen entwerfen und verwenden.
Jedes Objekt hat einen Verweis auf sein Klassen-Metaobjekt; Der Zugriff darauf erfolgt mit der Methode `class-of`.

`class-of`	`liefert das Klassen-Metobjekt des übergeben`

`Die folgende Beispiel-Funktion gibt den Namen der Klasse eines Objekts aus. Di`
`Objekts muß eine Subklasse von standard-object` sein (dies ist, falls nicht per MOP geändert, immer der Fall).

```
;; Der Parameter object ist vom "Typ" standard-object
;; Der "Typ" von stream ist nicht spezifiziert
(defun print-class-name ((object standard-object) stream)
    (format stream "~"
;;      rufe die Methode class-name für das gelieferte Klassenobjekt auf
        (class-name (class-of object))))
```

Eine Downcasting-Operation ist nicht erforderlich, da CLOS keine streng getypte Sprache ist.

5.3.3 Abfragen

5.3.3.1 Die Klassen-Metaklasse `standard-class`

Folgende Methoden sind unter anderem für die Metaklasse `standard-class` definiert:

`class-name`	`gibt den Klassennamen zurück`
`class-direct-superclasses`	`gibt eine Liste der Klassen-Metaobjekte der Superklassen zurück`
`class-direct-subclasses`	`gibt eine Liste der Klassen-Metaobjekte der Subklassen zurück`

[1] Dieser Fehler läßt sich natürlich mit Hilfe des MOPs beheben: dazu müßte man die Methoden `compute-slots, make-effective-slot-definition` und einige weitere Methoden, die die Vererbung von Slots bestimmen, überschreiben.

class-direct-slots	gibt eine Liste der Slot-Metaobjekte der in dieser Klasse definierten Slots zurück
class-slots	gibt eine Liste der Slot-Metaobjekte aller Slots dieser Klasse zurück (einschließlich der von Superklassen geerbten Slots)

Obwohl die Generic Functions nicht direkt mit den Klassen verbunden sind, gibt es eine Routine, die Auskunft über die relevanten Generic Functions gibt:

(relevant-generic-functions class)	gibt eine Liste aller Generic Functions zur der Klasse class als erstes Argument nehmen

5.3.3.2 Die Slot-Metaklasse slot-definition

Die Slot-Metaklasse slot-definition liefert folgende Meta-Information:

slot-definition-name	liefert den Namen des Slots
slot-definition-initargs	liefert die Namen der Parameter, die zur In Slots übergeben werden können
slot-definition-initform	gibt den Default-Initialisierungswert zurüc verwendet, falls bei der Erzeugung eines Ob Initialisierungsparameter für diesen Slot ü
slot-definition-location	gibt die Speicheradresse eines Slots an. Mi Methode kann man direkt auf den Inhalt eine

5.3.3.3 Die Generic-Function- und Methoden-Metaklassen

Zum Methodenaufruf werden zwei Arten von Metaobjekten verwendet: Ein Generic-Function-Metaobjekt (Metaklasse standard-generic-function) repräsentiert eine Menge von Methoden, die denselben Namen und die gleiche Lambda-Liste haben (d.h. dieselbe Anzahl Parameter und dieselben Namen für die Parameter). Ein Methoden-Metaobjekt (Metaklasse standard-method) repräsentiert genau eine Methode.
Generic-Function-Metaobjekte sind daher im wesentlichen nur Verwalter einer Liste von Methoden-Metaobjekten; die eigentliche Information (Signatur, Rumpf der Methode, usw.) ist in den Methoden-Metaobjekten enthalten. Demzufolge hat die Generic-Function-Metaklasse nur eine interessante Methode:

generic-function-methods	liefert die Liste aller Method-Metaobjekte Function zurück

Die Method-Metaklasse hat folgende wichtigen Methoden:

method-specializers	liefert eine Liste der Klassen-Metaobjekte meter zurück, mit denen diese Methode dekla
method-body	liefert den Methoden-Rumpf zurück
method-qualifiers	liefert Kennungen zurück, die eine Sonderbe Methodenaufruf erfordern, z.B. :before, :after

5.3.4 Object-I/O

Zum direkten Lesen und Schreiben von Slots dienen folgende Methoden:

(slot-value instance slot-name)	liefert den Wert des Slots slot-name des Objekts instance zurück
(setf (slot-value instance slot-name) new-value)	setzt den Slot slot-name des Objekts instance auf new-value

Die nötige Information, um über alle Slots eines Objekts zu iterieren, liefert der Methode class-slots.

5.3.5 Dynamischer Methodenaufruf

In einer LISP-artigen Sprache ist der dynamische Methodenaufruf natürlich eine der leichtesten Übungen. Dazu gibt es die Routine (apply-generic-function gf args). Diese Routine

ruft die Generic Function mit Namen `gf` mit den Argumenten `args` auf. Dabei wird die Polymorphie berücksichtigt, d.h. je nach Klassenzugehörigkeit der Argumente wird die entsprechende Methode aufgerufen. Beispiel:

```
(apply-generic-function #'paint (make-instance 'color-rectangle))
```

ruft die `paint`-Methode für das neu erzeugte `color-rectangle`-Objekt auf.

5.3.6 Eingriff in die Speicherung der Exemplarvariablen

Jeder Zugriff auf die Slots einer Klasse erfolgt über die Methoden `slot-value-using-class` zum Lesen und `(setf slot-value-using-class)` zum Schreiben, die in der Metaklasse `standard-class` definiert sind. Um in die Speicherung von Slots einzugreifen, muß man eine eigene Klassen-Metaklasse von `standard-class` ableiten und die obigen Methoden überschreiben. Klassen mit der neuen Metaklasse haben automatisch den geänderten Slot-Zugriff.

Wie das Verfahren konkret funktioniert, sei am Beispiel einer Metaklasse gezeigt, die automatisch Zugriffe auf die Slots der Exemplare ihrer Klassen mitprotokolliert. Zunächst benötigt man eine neue Klassen-Metaklasse, die von der Standard-Metaklasse erbt:

```
(defclass monitored-class (standard-class) () )
```

Zwei neue `before`-Methoden – eine zum Lesen, eine zum Schreiben – protokollieren alle Zugriffe für Klassen, die als Metaklasse `monitored-class` haben:

```
(defmethod slot-value-using-class :before
    ((class monitored-class) instance slot-name)
    (note-operation instance slot-name 'slot-value)

(defmethod (setf slot-value-using-class) :before
    (new-value (class monitored-class) instance slot-name)
    (note-operation instance slot-name 'set-slot-value)
```

Dazu braucht man nur noch eine Liste, in der die Zugriffe gespeichert werden, die Funktion `note-operation`, die einen Zugriff in die Liste einträgt, eine Reset- und eine Ausgabefunktion:

```
(let ((history-list) ()))

(defun note-operation (instance slot-name operation)
    (push '(,operation ,instance ,slot-name) history-list) (values))

(defun reset-slot-access-history ()
    (setq history-list ()) (values))

(defun slot-access-history ()
    (reverse history-list))
```

Ein Probelauf ergibt folgende Ausgabe (mit >> markiert):

```
defclass foo ()
    ((slot1 :accessor foo-slot1 :initarg :slot1)
    ((slot2 :accessor foo-slot2 :initform 200)
    (:metaclass monitored-class))

(reset-slot-access-history)
(setq i (make-instance 'foo :slot1 100))
>> #<FOO 381312>
(setf (slot-value i 'slot1) (foo-slot2 i))
>> 200
(incf (foo-slot1 i))
>> 201
(slot-access-history)
>> ((SET-SLOT-VALUE #<FOO 381312> SLOT1)        ;from initialization
>>  (SET-SLOT-VALUE #<FOO 381312> SLOT2)    ;from initialization
>>  (SLOT-VALUE #<FOO 381312> SLOT2)            ;from foo-slot2
>>  (SET-SLOT-VALUE #<FOO 381312> SLOT1)        ;from setf
>>  (SLOT-VALUE #<FOO 381312> SLOT1)            ;from incf (reading)
>>  (SET-SLOT-VALUE #<FOO 381312> SLOT1)        ;from incf (writing)
```

5.3.7 Methodenumlenkung

Zur Methodenumlenkung gibt es in CLOS spezielle Sprachkonstrukte:

- `before`-Methoden werden vor jedem Methodenaufruf aufgerufen.
- `after`-Methoden werden nach jedem Methodenaufruf aufgerufen.
- `around`-Methoden erhalten vor jedem Methodenaufruf die Kontrolle. Sie können mit `next-method` die Kontrolle an die ursprünglich gerufene Methode abgeben. Ist diese beendet, geht die Kontrolle wieder an die `around`-Methode zurück. Die Verschachtelung mehrerer solcher Methoden ist erlaubt.

Hier ein Beispiel mit den Klassen `color-mixin`, `rectangle` und `circle`:

```
(defmethod paint (shape color-mixin) :before
    (set-brush-color (color shape))
(defmethod paint (shape rectangle)
    (...)   ;;paint rectangle
(defmethod paint (shape circle)
    (...)       ;;paint circle

(defclass color-circle (circle color-mixin) ... )
```

Ein Aufruf von `paint` mit einem Exemplar von `color-circle` als Argument führt dazu, daß zunächst die `before`-Methode ausgeführt wird, die die Farbe setzt und danach erst die `paint`-Methode für Kreise.

5.3.8 Reflexion

Wie in Smalltalk ist auch in CLOS der Interpreter Teil des Laufzeitsystems. Der Entwickler schreibt neue Klassen, Methoden, usw. durch Aufruf von Methoden (oder Makros) des MOP. Diese Schnittstelle kann aber natürlich auch von einem laufenden Programm zur Selbstmodifikation oder -erweiterung genutzt werden. Folgendes sind die einschlägigen Makros:

(defclass name direct-superclasses direct-slots &rest options)	erzeugt eine neue Klasse mit Namen `name`, den Superklassen `direct-superclasses` und den Slots `direct-slots`. Es können noch Optionen angegeben werden; die wichtigste Option ist die angabe `:metaclass`, ein Exemplar welcher Metaklasse die neue
(defgeneric function-n lamb &res	erzeugt eine neue Generic Function mit dem `function-name` und den im `lambda-list` angegebenen formalen Parametern
(defmethod function-na qual para [1] body	erzeugt eine neue Methode mit dem Namen `function-name`. In `qualifiers` können Optionen übergeben z.B. `:before`. Die in `parameter` übergebene Parameterliste besteht aus Paaren von Par Parameterklasse. Die Angabe der Klasse de Parameters dient zum polymorphen Methoden `body` wird die Implementierung der Methode Liste (wie in LISP üblich) übergeben.

CLOS hat von LISP die Eigenschaft übernommen, daß Methoden nichts anderes als L
Programme können also ohne weiteres neue Methoden generieren. Mit Hilfe der v
Schnittstelle können natürlich auch Klassen automatisch generiert oder modifizie
Die Möglichkeiten, die Sprache zu verändern, werden in dieser Arbeit nicht vorge

5.3.9 Diskussion

CLOS bietet alles, was das Herz an MOP begehrt. Die Schnittstelle ist im Gegensatz zur Schnittstelle, die Smalltalk bietet, sehr gut verständlich. Das liegt auch an der vorbildlichen

[1] Die Deklaration lautet eigentlich `(defmethod &rest args)`. Die Argumente werden von einer eigenen Methode `parse-defmethod` ausgewertet.

vollständig objektorientierten Modellierung mit Klassen-Metaklassen, Methoden-Metaklassen und Variablen-Metaklassen. Diese Design führt dazu, daß der Entwickler mit wenig Aufwand eigene MOP-Funktionalität realisieren kann.
Leider besteht nur eine lose Kopplung zwischen „Objekten/Klassen“ und „Methoden“. Daher ist die Zustandskapselung in CLOS nicht gewährleistet.

6 Meta-Information in C++ - bisherige Ansätze

6.1 Das „Objektmodell“ von C++

C++ bietet dem MOP-Entwickler aus folgenden Gründen Schwierigkeiten:

- Nicht jede Variable ist ein Objekt; es gibt auch sog. Built-In-Datentypen, denen keine Klasse zugeordnet ist (`int`, `char`, `double`, usw.).
- Nicht einmal jeder strukturierte Datentyp ist eine Klasse.
- Es gibt das `union`-Konstrukt.
- Nicht jede Prozedur ist eine Methode.

Mit den letzten drei Punkten kann der MOP-Entwickler „kurzen Prozeß machen“, indem er einfach keine Meta-Information für `structs`, `unions` und Funktionen erzeugt. Allerdings dürfen dann Klassen keine Variablen eines `struct`- oder `union`-Datentyps haben.
Für Built-In-Datentypen jedoch muß jedes MOP eine Sonderfall-Behandlung vorsehen, da die Definition der Exemplarvariablen einer Klasse letztlich auf Built-In-Datentypen zurückführbar sein muß. Hier behilft man sich mit in der MOP-Bibliothek vordefinierten Metaobjekten und/oder baut eine entsprechende `switch`-Anweisung in die Methoden ein, die Exemplarvariablen behandeln(z.B. Object-I/O).

6.2 Exkurs: Die „Philosophie“ von C++[1]

C++ ist als Sprache für die Systemprogrammierung entstanden. Sowohl von der Geschichte her als auch vom jetzigen Zustand her ist es eine Erweiterung von C. Die Designprinzipien von C wurden beibehalten:

- Die systemnahe Programmierung soll möglich bleiben; es „darf keinen Platz für eine Sprache zwischen C++ und Assembler geben“.
- Umsetzung der Sprachkonstrukte in Maschinencode/Speicherlayout soll für den Programmierer leicht nachvollziehbar bleiben und die Runtime-Kosten für jedes Feature sollen leicht abschätzbar sein.
- Sprachfeatures, die nicht benutzt werden, sollen auch auch nichts kosten („zero overhead rule“).
- Die Syntax soll kurz und knapp sein.
- Die Sprache soll möglichst klein bleiben; statt dessen gibt es eine umfangreiche Standardbibliothek.

Zu diesen Grundsätzen kamen einige neue Designprinzipien für C++:

- Jedes Quelltextfile muß unabhängig von allen anderen übersetzbar sein; gelinkt wird mit einem Standard-Linker. Diese Forderung macht es etwas schwieriger, programmglobale Informationen, wie z.B. einen Behälter aller Metaklassenobjekte, zu Verfügung zu stellen. Es gibt jedoch Techniken, die auch dies erlauben.
- Soweit wie möglich ist die Kompatibilität zu C aufrechtzuerhalten.
- C++ ist eine Programmiersprache, kein vollständiges System wie Smalltalk. Daher sind Features wie Object-I/O o.ä. nicht Bestandteil von C++.
- Statische Typprüfung muß möglich sein; es darf keine implizite Verletzung des Typsystems geben[2].

Ein MOP für C++ sollte die obigen Grundsätze soweit wie möglich einhalten. Die Tatsache, daß jedes MOP das Typsystem verletzt (s. 3.2.2), hat Stroustrup lange davon abgehalten, ein MOP in C++ einzuführen, da er fürchtete, daß dadurch ein Programmierstil gefördert würde, der

[1] Diese Abschnitt basiert auf [Stroustrup94]

[2] Dieser Grundsatz hat inzwischen auch Eingang in die Entwicklung von C gefunden: Viele früher implizite Casts müssen in ANSI C explizit gemacht werden, es findet standardmäßig eine Typprüfung der Parameter bei Funktionsaufrufen statt.

Typprüfungen zur Laufzeit nutzlos macht. Da ein MOP für Frameworks jedoch zwingend erforderlich ist, hat Stroustrup inzwischen seine Meinung geändert und bei der Einführung eines (Mini-)MOP in C++ mitgearbeitet.

6.3 Der (zukünftige) C++-Sprachstandard

Bestandteil des zukünftigen C++-Standards wird ein MOP sein, das auf Klassen mit virtuellen Methoden beschränkt ist und folgende Funktionalität bietet [WP95]:

- eine Mini-Klassen-Metaklasse (die Klasse `type_info`);
- einen Operator `typeid()`, der das Klassen-Metaobjekt eines Objekts oder einer Klasse liefert;
- einen Operator `dynamic_cast<T>()`, mit dem man einen sicheren Downcast durchführen kann.

6.3.1 RTTI

Der `typeid()`-Operator erlaubt den Zugriff auf die `type_info`-Objekte. Er läßt sich mit einem Klassennamen, oder einem Ausdruck als Parameter aufrufen. `typeid( tSample )` liefert das Metaobjekt der Klasse `tSample` zurück, `typeid( Obj )` das Metaobjekt des dynamischen Typs von Obj.

Das Klassen-Metaobjekt ist im wesentlichen eine eindeutige Klassen-Id: Objekte derselben Klasse liefern immer dasselbe `type_info`-Exemplar zurück, Objekte unterschiedlicher Klassen liefern garantiert verschiedene `type_info`-Exemplare zurück. Außer der Vergleichbarkeit bietet das type_info-Objekt nur noch die Möglichkeit, sich den Klassennamen als String geben zu lassen.
Die Deklaration der Klasse `type_info` sieht folgendermaßen aus:

```
class type_info
{
    // implementation-dependent representation
    private:
    type_info( const type_info& )       // nicht kopierbar
    type_info& operator=(const type_info&);

    public:
    virtual ~type_info();

    int operator==( const type_info&);  // Vergleichbar
    int operator!=( const type_info&);

    int before( const type_info& ) const;   // sortierbar

    const char* name() const;       // liefert Klassennamen
};
```

6.3.2 Downcasting

Mittels des Operators `dynamic_cast<T>()` lassen sich sichere Downcasts durchführen. Bei Erfolg des Downcasts wird der gewünschte Zeigertyp zurückgeliefert, sonst der Null-Zeiger. Der Operator wird folgendermaßen benutzt:

```
Sample* s1 = new Sample();
Base* b1 = new Base();

Base* b2 = s1;

Sample* s2 = dynamic_cast<Sample*>( s1 );   // jetzt ist s2 = s1
s2 = dynamic_cast<Sample*>( b1 );       // jetzt ist s2 = 0
```

6.4 Metaflex

Metaflex ist wie MopGen ein Precompiler und arbeitet sehr ähnlich ([JohPal93]). Der MetaFlex-Code-Generator parst den Applikationsquelltext und erzeugt zusätzlichen C++-Quelltext, der übersetzt und zur Applikation hinzugelinkt wird. Die Besonderheit von Metaflex besteht darin, daß für jede Klasse spezifiziert werden kann, welche Teile des MOP erzeugt werden sollen. Dadurch verringern sich die Code-Größe und der benötigte Speicherplatz erheblich.

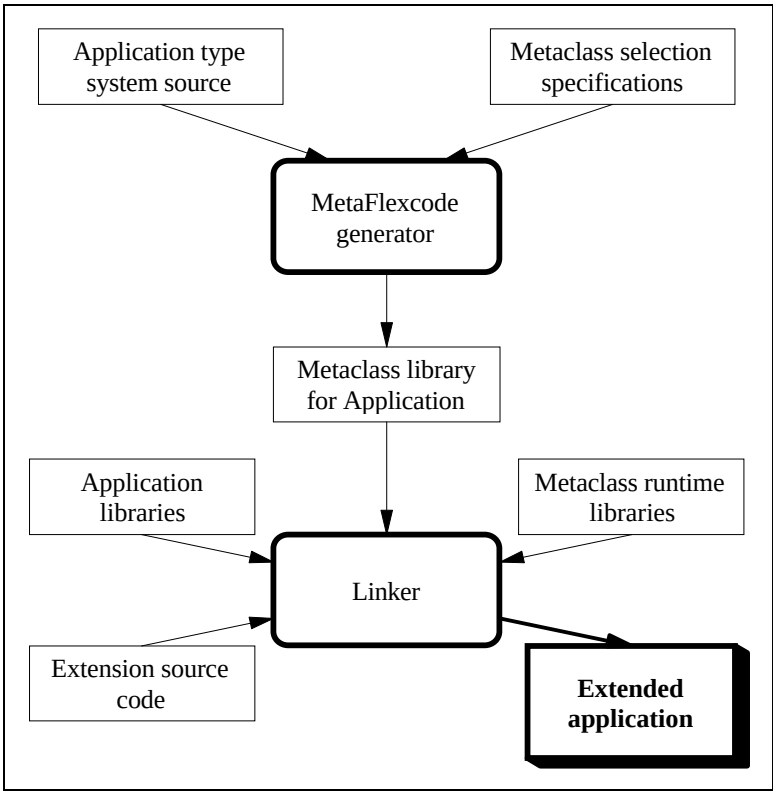

Diagramm 6.1 Überblick über den Übersetzungsprozeß mit MetaFlex

6.4.1 Die Funktionalität

Metaflex bietet als Standardfunktionalität eine Metaklasse, die u.a. dynamische Objekterzeugung, dyn. Abfrage der Klasse eines Objekts und Information über die Größe eines Exemplars enthält (genauere Angaben machen die Autoren leider nicht).
An diese Metaklasse können zusätzliche Meta-Info-Objekte angehängt werden, die weitere Funktionalität zur Verfügung stellen. Diese Meta-Info-Objekte werden ebenfalls von Metaflex erzeugt. Bisher stehen Meta-Info-Objekte für folgende Features zur Verfügung:

- Function Dictionary (Klassenschnittstellen-Information) (in erster Linie für Scripting; der Grund für die Entwicklung von Metaflex war der Wunsch nach Scripting-Unterstützung); Wichtigstes Feature: Dynamischer Methodenaufruf.
- Field Dictionary (Information über Exemplarvariablen)
- Instance Collections (Exemplarbehälter)
- Dynamc Createion (Erzeugung über Klasssennamen als String)

6.4.2 Die Konfigurationssprache

Die Besonderheit von Metaflex besteht darin, daß für jede Klasse ausgewählt werden kann, welche Teile des MOP erzeugt werden sollen. Dazu haben Johnson/Palaniappan eine eigene Spezifikationssprache entwickelt. Typische Spezifikationen sehen folgendermaßen aus:

```
MakeMetaClassFor    Class Eventhandler
```

```
MakeMetaClassFor    AllDerivedClassesOf Command
    Support FunctionDictionary
            Style FunctionDispatching
            AllowAccessTo PublicMembers
```

Diese Spezifikation gilt für alle Klassen, die von `Command` erben. Metaflex soll für diese Klassen ein `FunctionDictionary`-MetaInfo-Objekt erzeugen. Die Spezifikation `Style Function-Dispatching` bedeutet, daß das Function Dictionary alle Informationen enthalten soll, die zum Aufrufen von Methoden über ihren Namen nötig sind. Diese Information soll nur für public Methoden erzeugt werden (`AllowAccessTo PublicMembers`).

6.4.3 Bisherige Implementierung des Generators

Die vorgeschlagene Spezifikationssprache ist zur Zeit noch nicht realisiert. Bisher orientiert sich Metaflex bei der Erzeugung des MOP daran, ob bestimmte Methodendeklarationen in der Klassendefinition vorkommen. Diese Deklarationen werden durch Makros erzeugt.
Metaflex kann bisher das Basis-MOP, den Code für dynamischen Methodenaufruf und den Code für Exemplarbehälter (instance collections) generieren.

6.5 ET++

6.5.1 Die Funktionalität

ET++ ist ein Application Framework, das an der ETH Zürich und am UBILAB der schweizerischen Bankgesellschaft entwickelt wurde ([Gamma92]). Es wird hier als Paradigma für alle MOPs vorgestellt, die auf dem C(++)-Preprozessor beruhen.
In ET++ erben fast alle Framework- und alle Anwendungsklassen (indirekt) von der Klasse `Object`. Mehrfachvererbung wird nicht unterstützt. Alle Klassen, die von Object erben, werden mit einem Klassendeskriptor versehen; er ist ein Exemplar der (Meta-)Klasse `Class`, der folgende Information enthält:

- den Klassennamen;
- die Größe eines Exemplars;
- eine Referenz auf den Klassendeskriptor der Basisklasse;
- den Dateinamen und die Zeilennummer der Definition und der Implementation der Klasse;
- ein prototypisches Exemplar der Klasse.

Die virtuelle Methode `IsA()` gibt den Klassendeskriptor zurück.
Eine weitere virtuelle Methode, die Methode `Members()` erlaubt den Zugriff auf die Exemplarvariablen eines Objekts. Mit ihrer Hilfe können Name, Typ und Position innerhalb des Objekts der Exemplarvariablen abgefragt werden.
Außerdem gibt es noch die Methode `PrintOn()` zur Ausgabe eines Objektes auf einen Stream und `ReadFrom()` zum Wiedereinlesen eines Objektes von einem Stream.

6.5.2 Die Realisierung

Die Definition und Implementation erfolgt mittels der C-Makros `MetaDef()` und `MetaImpl()`. Hier ein Beispiel:

```
class ConnectionShape : public Shape
{
    enum{ red, blue, green } color;
    Shape* startShape;
    Shape* endShape;
    char* description;

    public:
    MetaDef( ConnectionShape );
    ConnectionShape();
    ...
};
```

In der Implementationsdatei müssen in einem Makro der Klassenname, der Basisklassenname und eine Liste der Exemplarvariablen angegeben werden:

```
MetaImpl( ConnectionShape, Shape,
        (T(color), TP(startShape), TP(endShape), TS(description), 0));
```

Die Exemplarvariablen werden je nach ihrem Typ von einem weiteren Makro umschlossen: `T()` für einfache Objekte, `TP()` für Zeiger, `TA()` für Arrays, usw.
Obwohl es schon die `Members()`-Methode zum Zugriff gibt, müssen für jede Klasse die Methoden `PrintOn()` und `ReadFrom()` handcodiert werden. Die Implementation für die obige Beispielklasse sieht folgendermaßen aus:

```
ostream& ConnectionShape::PrintOn( ostream& s )
{
    Shape::PrintOn( s );
    s << color << " " << startShape << " " << endShape;
    PrintString( s, description );
    return s;
}

istream& ConnectionShape::ReadFrom( istream& s )
{
    Shape::ReadFrom( s );
    s >> Enum( color ) >> startShape >> endShape;
    ReadString( s, &description );
    return s;
}
```

Es gibt zwei Gründe dafür, daß `PrintOn()` und `ReadFrom()` nicht auf `Members()` aufbauen:

- Erstens muß man oft nicht alle Exemplarvariablen abspeichern, um den Zustand eines Objekts wieder rekonstruieren zu können.
- Zweitens ist die Performance bei „hardcodierten" Ein- und Ausgaberoutinen besser als bei Methoden, die dazu Schleifenkonstrukte abarbeiten und Abfragen über Typ und Größe der Exemplarvariablen durchführen müssen.

6.6 Grossman's Parser Code Generator

6.6.1 Die Funktionalität

Grossman's MOP-Generator [Großman93] hat im wesentlichen die Aufgabe, Code für Object-I/O zu erzeugen. Um die in C++-Streams abgespeicherten Objekte auch wieder laden zu können, benötigt man RTTI, die ebenfalls vom MOP-Generator erzeugt wird.
Für welche Klassen Code erzeugt werden soll, kann der Entwickler mittels eines `#pragma` festlegen. Soll z.B. die Klasse `Sample` persistent werden, muß man vor der Klassendeklaration den Text `#pragma persistence declaration Sample` einfügen.
Vom MOP-Generator werden neue Header-Files erzeugt, die von jedem C++-Compiler übersetzt werden können. Die erzeugten Header-Files enthalten nicht nur die Deklaration der hinzugefügten Methoden, sondern auch ihre Implementierung.

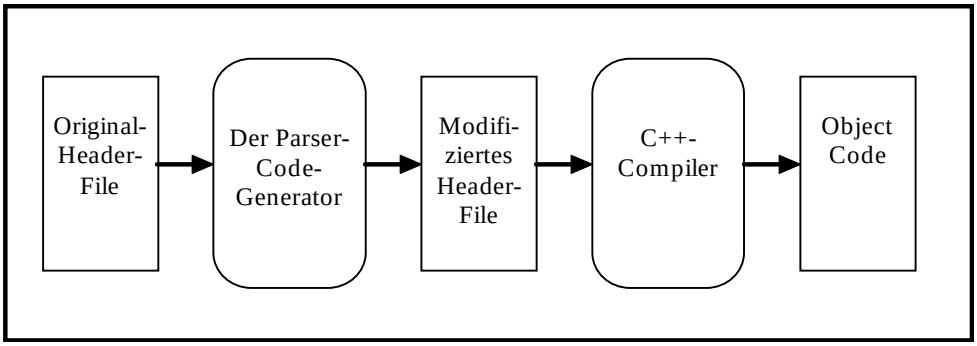

Diagramm 6.2 Der Parser Code Generator

6.6.2 Die Realisierung des Object-I/O

Das Speichern eines Objekts erfolgt mit Hilfe der Methode `flattenEntity()`. Da die entsprechenden Basisklassen-Routinen zum Speichern der Exemplarvariablen aufgerufen werden müssen, ruft `flattenEntity()` einen speziellen Speicherungskonstruktor auf. Dadurch, daß es ein Konstruktor ist, werden automatisch die passenden Speicherungskonstruktoren der Basisklassen aufgerufen. Der Konstruktor zum Speichern wird nur seiner Seiteneffekte wegen aufgerufen. Das erzeugte Objekt wird sofort nach dem Speichern wieder gelöscht. Der Speicherungskonstruktor ruft für alle Objekte, auf die das zu speichernde Objekt Referenzen enthält, wieder `flattenEntity()` auf. Damit es nicht zur Endlosrekursion kommt, wird in einer Tabelle `SaveEntityMap` festgehalten, welche Objekte schon abgespeichert worden sind.
Jede Klasse erhält eine Klassen-Id. Diese Id wird beim Speichern des Objekts zusammen mit dem Objekt abgelegt.
Beim Start des Programms wird eine Tabelle erzeugt, die für jede Klassen-Id einen Zeiger auf eine Funktion `makeNew` enthält, die ein neues Objekt dieser Klasse erzeugt und die gespeicherten Daten einliest (dynamic creation). Das Einlesen eines Objekts geschieht mit der Klasenmethode `unflattenEntity(), die das neue Objekt zurückliefert`. Beim Einlesen eines Objekts wird zunächst die Klassen-Id gelesen und dann die entsprechende `makeNew`-Funktion aufgerufen. Diese ruft einen speziellen Einlese-Konstruktor auf, der die Exemplarvariablen vom Stream liest. Dabei erfolgen ggf. analog zum Schreiben rekursive Aufrufe von `unflattenEntity()`. Wie beim Schreiben gibt es eine Liste der bereits gelesenen Objekte, ReadEntityMap. Aus technischen Gründen wird ein neues Objekt nicht von `unflattenEntity()`, sondern von einem eigenen `new`-Operator in die `ReadEntityMap` eingetragen.
Der Generator muß also folgende Methoden für jede persistente Klasse erzeugen:

```
// Klassen-Id:
static MG_ClassId classId;

// Methoden zur Ausgabe:
virtual const void* flattenEntity(MG_baseFormatOut& strm ) const;
Sample( MG_baseFormatOut& strm, const Id& theObject )

// Methoden zum Einlesen:
static Sample* unflattenEntity( MG_baseFormatIn& strm);
Sample( MG_baseFormatIn& )
static void* makeNew( anObjectId id, MG_baseFormatIn& strm );
operator new ( size_t size, anObjectId id );
```

Diese Realisierung funktioniert nicht mit Zeigern auf eingebettete Objekte; die Benutzung solcher Zeiger ist daher nicht erlaubt.

6.7 Das Metadata System

6.7.1 Die Funktionalität

Wesentliche Anwendung des Metadata Systems ist „die Funktionalität, zur Laufzeit [den Zustand der] Objekte zu betrachten und zu ändern“[PWJ92]. Daher liegt der Schwerpunkt der zur Verfügung gestellten Information auf den Exemplarvariablen. Man kann auch die Basisklasse einer Klasse abfragen, aber keine Information über die Art der Vererbungsbeziehung (public/protected/private, virtual). Mehrfachvererbung ist nicht erlaubt. Informationen über Methoden werden nicht zur Verfügung gestellt.
Die Stärke des Metadata Systems liegt darin, daß die komplette Information mit Hilfe von Metaklassen modelliert wird (anstatt z.B. durch zusätzliche Methoden). Der Klassenschnittstelle werden nur Zugriffsmethoden auf das Metaobjekt hinzugefügt:

```
class Example
{
  private:
    static VMeta* cvPVMeta;     // Zeiger auf das Klassen-Metaobjekt
    ...
  public:
    virtual VMeta* GetVMeta();
    static  VMeta* GetVMetaObj();
    ...
};
```

Es werden nicht nur für Klassen, sondern auch für mit typedef definierte Zeigertypen, Aufzählungstypen, usw. Metaobjekte erzeugt. Für die Builtin-Typen gibt es vorgefertigte Metaobjekte. Außerdem gibt es eigene Variablen-Metaobjekte, die die Information über Exemplarvariablen zur Verfügung stellen, und eigene Aufzählungskonstanten-Metaobjekte.
Folgende Informationen werden zur Verfügung gestellt:

In den Typ-Metaobjekten:

- Name
- Referenz auf Typ-Metaobjekt der Basisklasse (falls vorhanden)
- Objektgröße
- Anzeigeformat (int, String, zusammengesetztes Format, usw.)
- für Aufzählungstypen: Array der Aufzählungskonstanten-Metaobjekte
- für Zeigertypen: Anzahl der Indirektionen (`int** hat 2 Indirektionen`)
- für Klassen: Array der Variablen-Metaobjekte für die Exemplarvariablen

In den Variablen-Metaobjekten:

- Name
- Typ
- Deklarationsart (public/protected/private)
- Anzahl der Zeigerindirektionen
- Für Arrays: Größe
- Offset zum Beginn des Objekts im Speicherlayout

In den Aufzählungkonstanten-Metaobjekten:

- Name
- Wert (als `int`)

Die Typ-Information ermöglicht die Standard-RTTI-Funktionalität. Mit Hilfe der Exemplarvariablen-Information lassen sich z.B. Browser entwickeln (ein Objekt-Browser für MS-Windows ist die erste Anwendung des Metadata Systems). Die Realisierung von Object-I/O scheitert daran, daß beim Einlesen Exemplare neu erzeugt werden müssen, deren Klasse nur als Zeichenkette oder Typ-Id bekannt ist; eine entsprechende Methode des Klassen-Metaobjekts wird nicht generiert. Ein weiterer Nachteil besteht darin, daß keine Mehrfachvererbung erlaubt ist.

6.8 Open C++

6.8.1 Die Funktionalität

Das MOP von Open C++ ([ChiMas93])beschränkt sich auf die Modifikation von Methodenaufrufen; es ist für die Entwicklung verteilter OO-Systeme gedacht. Methodenaufrufe werden nicht mehr direkt ausgeführt, sondern zunächst an ein Metaobjekt weitergereicht, das seinerseits die ursprüngliche Methode aufruft. Das Metaobjekt kann vor und nach dem Aufruf weitere Aktionen durchführen, den Methodenaufruf an ein anderes Objekt weiterleiten (für verteilte Objekte), usw.

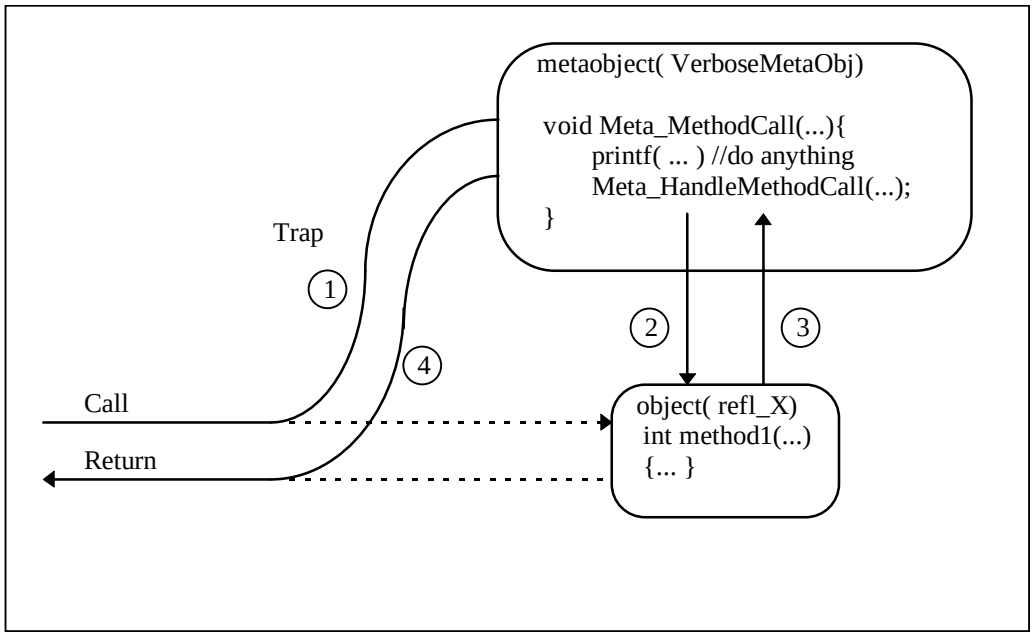

Diagramm 6.3 Die Methodenaufruf-Umlenkung in Open C++

Im abgebildeten Beispiel wird der Aufruf der Methode refl_X::`method1()` umgelenkt. Zunächst wird die Methode `Meta_MethodCall()` aufgerufen, die nun beliebige Operationen durchführen kann. Im obigen Beispiel wird der Methodenaufruf mitprotokolliert. Danach kann entweder die Original-Methode aufgerufen werden (mit `Meta_HandleMethodCall()`) oder das Metaobjekt kann den Methodenaufruf an ein anderes Metaobjekt weiterleiten. Letzteres ist im Fall verteilter Objekte oft sinnvoll.

6.8.2 Die Realisierung der Methodenumlenkung

Die Methodenumlenkung wird realisiert, indem von jeder benutzerdefinierten Klasse X, die mit Methodenaufrufumlenkung ausgestattet werden soll, eine Klasse refl_X abgeleitet wird. Diese Klasse wird automatisch von einem Code-Generator erzeugt. Um die MOP-Funktionalität zu nutzen, muß der Entwickler Objekte der neuen Klasse refl_X anstelle von X verwenden. Mittels Kommentaren kann der Entwickler festlegen, welche Klassen und welche Methoden die neue Funktionalität erhalten sollen. Das Vorgehen wird an folgendem Beispiel deutlich:

```
class X
{
    public:
    X();
//MOP reflect:
    int method1( int par1, int par2);
    long method2( long par1 );
};

//MOP reflect class X : VerboseMetaObject
```

Der letzte Kommentar weist den Generator an, die Klasse X mit einem Metaobjekt der Klasse `VerboseMetaObject` zu verbinden. Von dieser Klasse ist hauptsächlich die Methode `Meta_MethodCall` interessant, die jeden Methodenaufruf mitprotokolliert:

```
class VerboseMetaObj : public MetaObj
{
    public:
    void Meta_MethodCall( Id method, Id category,
                          ArgPac& args, ArgPack& reply )
    {
        printf( "method %s was called.\n", Meta_GetMethodName( method ) );
        Meta_HandleMethodCall( method, args, reply );
    }
};
```

Alle Metaklassen erben von der Klasse MetaObj. Der Entwickler kann selbst neue Metaklasen entwickeln.
Die generierte Klasse `refl_X` sieht folgendermaßen aus[1]:

```
class refl_X : public X, public MetaObj
{
    static MethodId idarray[];

    int method1( int par1, int par2 );
    long method2( long par1 );
    virtual void method1_call( ArgPac& args, RetPac& reply )
    virtual void method2_call( ArgPac& args, RetPac& reply )
};

static MethodId refl_X::idarray[] =
{
    { "method1", 1, &method1_call },
    { "method2", 2, &method2_call }
}

int refl_X::method1( int par1, int par2 )
{
    ArgPac args;
    args.add( &par1 );
    args.add( &par2 );
    RetPac reply;
    Meta_MethodCall( idarray[1], StdMethod, args, reply );
    return reply.GetAsInt();
}

void refl_X::method1_call( ArgPac& args, RetPac& reply )
{
    reply.SetAsInt( X::method1(
                        args[1].GetAsInt(),
                        args[2].GetAsInt() ) );
}

... // das gleiche für method2
```

Die folgenden beiden Methoden werden nicht von `refl_X` überschrieben:

```
VerboseMetaObj::Meta_MethodCall(
                MethodId& id, Id category, ArgPac& args, RetPac& reply )
{
    printf( "method %s was called.\n", Meta_GetMethodName( method ) );
    Meta_HandleMethodCall( method, args, reply );
}

MetaObj::Meta_HandleMethodCall(
                MethodId& id, Id category, ArgPac& args, RetPac& reply )
{
    id.Method( args, reply );
}
```

Der Aufruf `id.Method()` ruft die ursprüngliche Methode auf.

[1] Die Autoren stellen die Implementation von refl_X nicht vor; die hier vorgestellte Realisierung paßt jedoch genau zu den im Artikel abgebildeten Code-Fragmenten der anderen Klassen und ist daher wohl der tatsächlichen Realisierung zumindest sehr ähnlich.

6.9 Das Meta-Information-Protocol

6.9.1 Die Funktionalität

Das Meta-Information-Protocol (MIP, [BKS92]) ist änlich aufgebaut wie das Metadata System (6.7). Es gibt nicht nur für Klassen, Built-In-Datentypen, Aufzählungstypen und Exemplarvariablen Metaobjekte, sondern auch für Methoden und Unions. Ein weiterer Unterschied besteht darin, daß die Typ-Information von Typen mit Typ-Modifizierer (z.B. `int*`) mit Hilfe verketteter Metaobjekte repräsentiert wird.

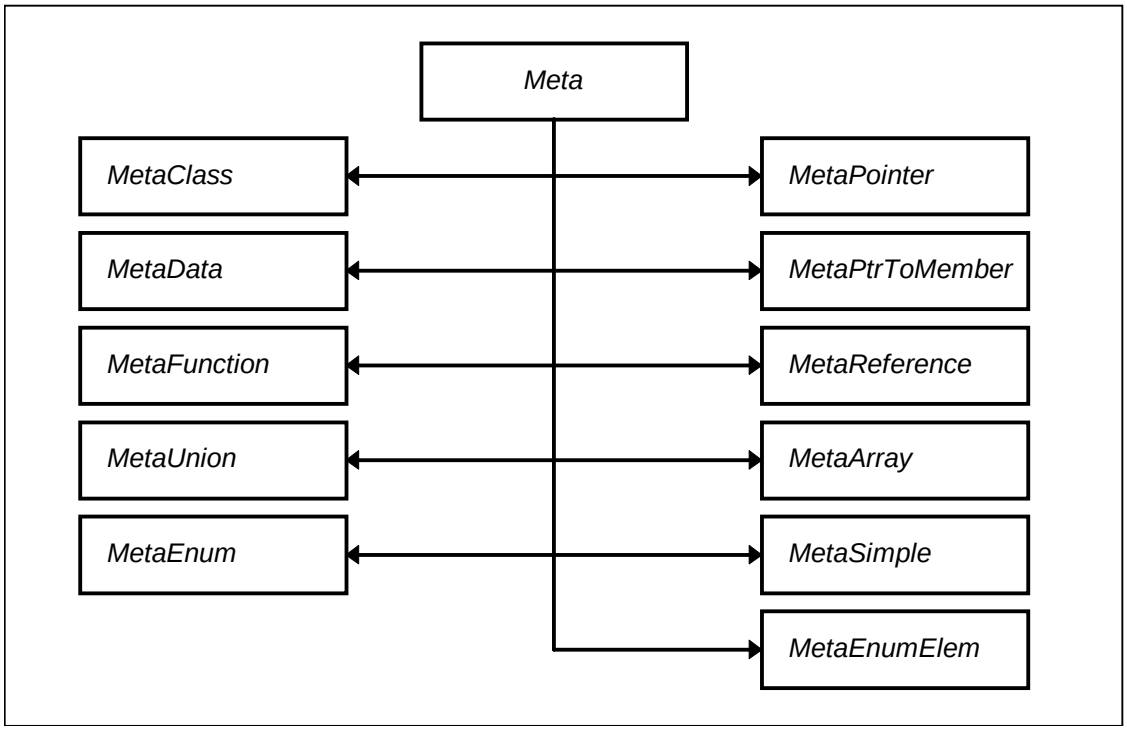

Diagramm 6.4 Die Metaklassen des MIP

Diese Metaobjekte werden von einem MIP-Generator erzeugt, der zwischen den C++-Preprozessor und den C++-Compiler geschaltet ist.
Die Klassen-Metaobjekte haben u.a. folgende Methoden:

name()	liefert den Klassennamen
size()	liefert die Größe eines Exemplars
baseClassIter()	iteriert über die Basisklassen. Rückgabewert ist das Metaobjekt der erreichten Basisklasse
dataMemberIter()	iteriert über die Exemplarvariablen und liefert das jeweilige Exemplarvariablen-Metaobjekt zurück
fctMemberIter()	iteriert über die Methoden und liefert das jeweilige Methoden-Metaobjekt zurück

Über die Methoden der anderen Metaklassen werden in [BKS92] nur fragmentarische Angaben gemacht. Ebensowenig wird über die Realisierung der Meta-Information berichtet. Daher ist es auch nicht möglich, eine MopSpec-Spezifikation für das MIP zu geben. Zusammenfassend kann man sagen, daß offenbar das Fehlen eines spezifischen Anwendungsschwerpunktes dazu geführt hat, daß die zur Verfügung gestellten Metaobjekte „in der Luft hängen“ und daher kaum praktisch anwendbar sind. Dieser Schluß muß jedenfalls aus der Tatsache gezogen werden, daß keinerlei Angaben darüber gemacht werden, wie mit Hilfe des MIP tatsächliche Funktionalität wie Object-I/O oder dynamischer Methodenaufruf realisiert werden können. Das einzige explizit erwähnte Feature, der Downcast, funktioniert, so wie er beschrieben ist, nicht mit Mehrfachvererbung und virtuellen Basisklassen.

6.10 Zusammenfassung und Diskussion

Die Vielfalt der Metaobjekt-Protokolle für C++ offenbart folgendes Dilemma: ein MOP, daß sich auf die Schnittmenge aller MOPs beschränkt, ist so klein, daß es niemandem ausreicht. Ein MOP dagegen, daß aus der Vereinigungsmenge aller realisierten MOPs besteht, ist so groß, daß es schon allein wegen seines Speicherplatzverbrauchs unrealistisch wird. Außerdem werden mit Sicherheit mit jedem größeren Framework neue Anforderungen an ein MOP hinzukommen. Der Einschätzung Stroustrups, daß kein realistisches MOP die Anforderungen aller Applikationen erfüllen kann, muß man sich wohl anschließen.
Hinzu kommt, daß von einer Vereinheitlichung eines C++-MOPs noch keine Rede sein kann. Bis auf eine kleine Schnittmenge (RTTI) ist noch nicht klar, welches Design das Beste ist. Oft hängt dies auch vom Schwerpunkt der Nutzung des MOP ab (DB-Zugriff, Scripting, Debugging). Daher ist es nötig, daß jeder Entwickler selbst entscheiden kann, wie sein MOP aussehen soll.
Er hat zur Zeit zwei Möglichkeiten zur Realisierung eines MOPs für C++: Entweder er realisiert es mit Hilfe von C-Preprozessor-Makros oder mit Hilfe eines eigenen Precompilers.

Die Makro-basierte Lösung hat folgende Vorteile:

- Die Implementierung des MOPs ist relativ unaufwendig (im Verhältnis zur Entwicklung eines Precompilers).
- Das MOP ist vollständig portabel.
- Für die Programmübersetzung ist keine zusätzliche Precompiler-Phase notwendig (kürzere turn-around-Zeiten).

Dies erkauft man allerdings mit folgenden Nachteilen:

- Die Implementierung ist bedingt durch die eingeschränkten Möglichkeiten des C-Preprozessors extrem kryptisch. Dadurch ist die Wartbarkeit oder Erweiterung durch den „einfachen“ Anwendungsentwickler praktisch nicht möglich.
- Aufgrund der engen Grenzen des C-Preprozessors sind viele Features praktisch nicht zu realisieren (z.B. Methodenumlenkung á la Open C++).
- Das MOP muß explizit vom Entwickler erzeugt werden, indem er für jede Klasse die entsprechenden Makroaufrufe in deb Quelltext einfügt. Abgesehen davon, daß dieses Vorgehen je nach Umfang des MOP etwas mühsam ist, hat es den großen Nachteil, daß die Meta-Information bei Änderung des Quelltextes nicht automatisch angepaßt wird. Hierfür ist ebenfalls der Entwickler zuständig.

Die Precompilerlösung bietet folgende Vorteile:

- Die Erzeugung des MOP und insbesondere die Anpassung an Quelltextänderungen geschieht automatisch.
- Diejenige Funktionalität, die für das Anwendungsgebiet wichtig ist, kann als implizite Funktionalität zur Verfügung gestellt werden, indem der MOP-Generator entsprechende Methoden generiert.
- Die generierten Methoden sind leichter verständlich als bei der Makro-basierten Lösung. Dies ist insbesondere beim Debugging von Vorteil.

Dafür handelt man sich andere Nachteile ein:

- Ein eigener MOP-Generator muß einen C++-Parser enthalten; die Implementierung ist daher aufwendig.
- Welche implizite Meta-Information generiert wird, ist in den bisherigen MOP-Generatoren „festverdrahtet“ und richtet sich nach dem geplanten Anwendungsschwerpunkt des MOP. Der Anwendungsentwickler hat hier keine Einwirkungsmöglichkeit (Metaflex bietet ihm wenigstens die Möglichkeit, zu entscheiden, welche Bestandteile des MOP für welche Klassen generiert werden sollen).
- Damit der Entwickler trotzdem flexibel bleibt, wird die gesamte Meta-Information zusätzlich als explizite Information zur Verfügung gestellt, mit allen Nachteilen, die explizite Meta-Information hat.

Sofern das benötigte MOP nicht allzu umfangreich ist, wiegen die Vorteile der Makro-basierten Lösung größer als ihre Nachteile. Ab einer bestimmten Größe wird ein Makro-basiertes MOP jedoch schwer zu warten. In diesem Fall ist der Precompiler-Lösung der Vorzug zu geben. Bis auf den Nachteil der längeren turn-around-Zeiten lassen sich die Nachteile der Precompilerlösung vermeiden, wenn der Entwickler selbst einen Einfluß darauf hat, welcher Code erzeugt werden soll. Der zweite Teil dieser Arbeit beschreibt MopGen, einen MOP-Generator, der genau diese Einflußmöglichkeit bietet.

II MopGen

7 MopGen: Überblick und Einordnung

7.1 Überblick

MopGen (Metaobjekt-Protokoll-Generator) bietet die Möglichkeit, die für die eigene Anwendung benötigte Meta-Information zu generieren. Sie wird als zusätzlicher C++-Quelltext erzeugt, der vom C++-Compiler übersetzt und zur Anwendung hinzugelinkt wird. Hierin unterscheidet sich MopGen nicht von den anderen Ansätzen für C++.
Die Besonderheit von MopGen besteht darin, daß er in der Flexibilität über die bisherigen Ansätze hinausgeht. Die einfacheren Metaobjekt-Protokolle sind für alle Klassen identisch (z.B. MIP); die moderneren bieten schon die Möglichkeit, auszuwählen, welcher Teil des MOPs für welche Klassen erzeugt werden soll (Metaflex); MopGen erlaubt es dem Anwendungsentwickler auf einfache Weise sein eigenes MOP zu entwerfen und zu implementieren, ohne daß er den Generator modifizieren muß.
Im Unterschied zu den oben vorgestellten Ansätzen für C++ bietet MopGen dem Anwendungsentwickler die Möglichkeit, mit Hilfe einer spezielle Metasprache zu spezifizieren, wie der zusätzlich erzeugte Quelltext aussehen soll. Diese Sprache wird in Kapitel 8 beschrieben.
Die gewünschte Spezifikation muß in einer Spezifikationsdatei angegeben werden. Zur Codeerzeugung liest MopGen sowohl die Deklarationen aller Klassen als auch die Spezifikationsdatei ein und erzeugt den gewünschten Quelltext.
Eine Beispiel-Spezifikation für ein Meta-Object-Protokoll ist vorhanden und kann als Ausgangspunkt für Erweiterungen verwendet werden. Die Verwendung dieses MOPs ist jedoch in keiner Weise zwingend. Anhand des Beispielmetaprotokolls wird die Verwendung der Metasprache erläutert (Kapitel 10). Das vorgestellte MOP ist nicht in jeder Hinsicht optimal. Es soll in erster Linie die Möglichkeiten von MopSpec aufzeigen und nicht als „Referenz“-MOP für C++ dienen.

7.2 Einordnung

Zur Einordnung von MopGen müssen zwei Ebenen unterschieden werden: die Ebene der MOP-Spezifikation (mit Hilfe von MopSpec) und die Ebene des Laufzeit-MOPs.
Im Gegensatz zu den oben aufgeführten Arbeiten liegt der Schwerpunkt dieser Arbeit darauf, dem Entwickler die Entwicklung eigener MOPs mit wenig Aufwand zu ermöglichen, anstatt das 42. möglichst universelle MOP zu implementieren. Daher ist die Ebene der MOP-Spezifikation hier die entscheidende. Die Spezifikation muß zur Übersetzungszeit erfolgen. Damit ist klar, daß MopSpec nur statische Meta-Information zur Verfügung stellen kann. Es wird nicht zwischen Klassen und Typen unterschieden. Zur Zeit wird die MOP-Generierung nur für Klassen unterstützt, nicht für andere C++-Typen wie Built-In’s, Enumerations, usw. Diese Einschränkung ist jedoch keine prinzipielle.
Mit Hilfe von MopSpec kann man Meta-Information und Klassen-Extensionen spezifizieren, nicht jedoch Sprach-Modifikationen.
Nun zur Ebene des Laufzeit-MOPs. Hier kann nur die Funktionalität des Beispiel-MOPs kurz umrissen und eingeordnet werden. Das Beispiel-MOP bietet nur die nötigste explizite Information, weil die meisten Verwendungen eines MOP sich viel besser als implizite Funktionen mit MopSpec spezifizieren lassen (s. als Beispiel das Object-I/O, Abschnitte 10.4 und 12.7).

8 MopSpec: Die Sprache zur Erzeugung von Meta-Information

Um jedem Entwickler die Möglichkeit zu bieten, sein eigenes dynamisches MOP zu definieren und zu implementieren, hat er über eine Art Meta-Makro-Sprache – MopSpec – den Zugrif auf alle statischen Informationen über die im Programm enthaltenen Klassen. MopGen liest den in MopSpec geschriebenen Code ein und erzeugt daraus für jede Klasse eine C++-Quelltextdatei mit dem generierten MOP.

Der MopSpec-Code ist praktisch eine Vorlage für den zu erzeugenden Quelltext mit eingestreuten Kontroll- und Generierungsanweisungen. Die mächtigsten Anweisungen sind die Kontrollanweisungen. Mit ihnen kann man abhängig von Bedingungen unterschiedlichen Quelltext generieren (`$If`...`$Else`-Anweisung) und Textteile mehrfach generieren (z.B. für jede Exemplarvariable einmal mit `$ForEachField`). Zur Steuerung der `$If`-Anweisung gibt es Abfrage-Anweisungen, die true oder false zurückliefern (z.B. liefert `$ClassIsA( C )` dann true zurück, wenn die aktuell bearbeitete Klasse typkompatibel zu C ist). Kontrollanweisungen beziehen sich immer auf den folgenden Text, der in geschweiften Klammern eingeschlossen sein muß.
Der mittels der Kontrollanweisungen ausgewählte Text wird in den neuen Quelltext kopiert; er kann natürlich weitere Kontrollanweisungen enthalten. Er kann aber auch sog. generative Anweisungen erhalten; diese Anweisungen werden im generierten Quelltext durch die entsprechende Information über die aktuell bearbeitete Klasse ersetzt (z.B. wird die Anweisung `$ClassName` durch den Namen der bearbeiteten Klasse ersetzt). Die meisten dieser Anweisungen sind nur in bestimmten Kontrollanweisungsblöcken erlaubt. So wird z.B. `$FieldName` durch den Namen der Exemplarvariablen ersetzt, für die gerade innerhalb einer `$ForEachField`-Anweisung der Quelltext generiert wird. Am folgenden Beispiel soll das Verfahren verdeutlicht werden.

8.1 Ein einführendes Beispiel: Objekt-Ausgabe

Als Beispiel sei eine Methode gewählt, die ein Objekt in einen Speicher schreibt. Es sollen nur diejenigen Exemplarvariablen ausgegeben werden, die nötig sind, um den Zustand des Objekts wiederherzustellen. Da dies nicht automatisch feststellbar ist, muß der Entwickler die entsprechenden Exemplarvariablen markieren. Dies kann er durch Setzen des `Persistent`-Flags:

```
class tGraphObject : public virtual tStorable {
    ...
};

class tRectangle : public virtual tGraphObject {
    SetFieldFlag( Persistent )
    double       x, y, w, h;
    tGraphObject* next;
    tGraphObject* parent;
    tString      descr;
    ResetFieldFlag( Persistent )

    tBitmap* buffer;

    public:
    virtual const tBitmap* GetAsBitmap() const;
    ...
};
```

In diesem Fall ist `buffer` die einzige Exemplarvariable, die nicht ausgegeben werden soll. Je nach Typ der Exemplarvariablen sollen verschiedene Methoden des Objekt-Speichers aufgerufen werden. Für unsere Beispielklasse `tRectangle` sieht die Ausgabemethode folgendermaßen aus

```
void tRectangle::WriteOnStore( tStore& store )
```

```
{
    tGraphObject::WriteOnStore( store ); // zunächst die Exemplarvariablen
                                         // der Basisklasse(n) ausgeben

    store.WriteBuiltIn( "x", x );        // Built-In Typen werden mit
    store.WriteBuiltIn( "y", y );        // WriteBuiltIn() ausgegeben
    store.WriteBuiltIn( "w", w );
    store.WriteBuiltIn( "h", h );
    store.WritePointer( "next", next );  // Zeiger mit WritePointer()
    store.WritePointer( "parent", parent );
    store.WriteObject( "descr", descr ); // und eingebettete Objekte mit
                                         // WriteObject()
}
```

Diese Methode soll für alle Klassen generiert werden, die direkt oder indirekt von `tStorable` abgeleitet sind. Der MopSpecCode, der diese Methode generiert, sieht folgendermaßen aus:

```
$If ( $ClassIsA( tStorable ) ) {                    // erzeuge Methode nur
für
                                                    // tStorable-Klassen
  void $ClassName::WriteOnStore( tStore& store )
  {
    $ForEachBase {                                  // für jede Basisklasse
        $If ( $BaseIsA( tStorable ) ) {
            $BaseName::WriteOnStore( store );       // WriteOnStore-Aufruf
                                                    // generieren
        }
        $Else {
            $Error( Alle Basisklassen müssen von tStorable erben )
        }
    }

    $ForEachField {                              // für alle Ex.Variablen
        $If ( $FieldFlagIsSet( Persistent ) ) { // nur Code erzeugen, wenn
                                                // Persistent-Flag gesetzt
            $If ( $FieldIsBuiltin ) {
                $If ( $FieldIsObject ) {
                  store.WriteBuiltIn( "$FieldName", // WriteBuiltIn()-Aufruf
                                      $FieldName );// generieren
                }
                $Else {
                  Error( Keine Zeiger, Ref., usw. auf Built-In's erlaubt)
                }
            }
            $Else {
                $If ( $FieldIsA( tStorable ) ) {
                    $If ( $FieldIsObject ) {
                      store.WriteObject( "$FieldName",
                                         $FieldName );
                    }
                    $Else $If ( $FieldIsPointer ) {
                      store.WritePointer( "$FieldName",
                                          $FieldName );
                    }
                    $Else {
                        $Error( Ex.Variable ist weder Objekt noch Zeiger )
                    }
                }
                $Else {
                    $Error( Ex.Variable ist weder Built-In noch
                            tStorable-typkompatibel )
                }
```

```
            } // $If ( ..Builtin )
          } // $If( ..Persistent )
        } // $ForEachField
      } // WriteOnStore()
    } // $If ( ...tStorable)
```

Es sei darauf hingewiesen, daß der MopSpec-Code für eine praxistaugliche Ausgabe-Methode noch etwas aufwendiger gestaltet werden muß. Z.B. behandelt dieser Beispiel-Code virtuelle Basisklassen nicht korrekt: sie werden u.U. mehrfach ausgegeben. Eine vollständige Spezifikation für brauchbares Object-I/O ist in Abschnitt 12.7 zu finden.

8.2 Die Anweisungen

8.2.1 Kontrollanweisungen

Es gibt folgende Kontrollanweisungen:

$If(<Expr>) { <Block1> } [$Else{ <Block2> }]	**Falls Expr True ist, wird Text1 ausgewertet. Falls Expr Fa ist und es einen $Else-Teil gibt, wird Text2 ausgewertet. Die Expr darf aus mit &And, $Or und $Not verknüpften Abfragen bestehen.**
$And, $Or, $Not	**Verknüpfungsoperatoren für die logischen Ausdrücke.**
$ForEachBase{ <Block> }	**<Block> wird für jede Basisklasse der aktuellen Klasse ausgewertet.**
$ForEachField{ <Block> }	**<Block> wird für jede Exemplar- oder Klassenvariable der aktuellen Klasse ausgewertet.**
$ForEachMethod{ <Block> }	**<Block> wird für jede Methode der aktuellen Klasse ausgewertet.**
$ForEachParam{ <Block> }	**<Block> wird für jeden Parameter der aktuellen Methode ausgewertet (nur innerhalb einer $ForEachMethod-Anweisung erlaubt).**

8.2.2 Klassenspezifische Anweisungen

Die MOP-Spezifikations-Datei wird zu Beginn der Generierung von MopGen eingelesen und dann für jede Klasse, deren Deklaration sich geändert hat[1], Code generiert. Alle $Class...-Anweisungen beziehen sich auf die Klasse, für die gerade Code generiert wird.

$ClassName	**fügt den Klassennamen in den generierten Code ein.**
$ClassDeclFile	**fügt den Namen der Datei ein, in der die Klasse deklariert wurde.**
$ClassIsA(<Class>)	**liefert True, wenn die Klasse typkompatibel zu Class ist.**

8.2.3 Basisklassenspezifische Anweisungen

Die folgenden Anweisungen sind nur innerhalb einer $ForEachBase-Anweisung erlaubt und beziehen sich auf die Basisklasse, für die gerade Code generiert wird.

$BaseName	**fügt den Namen der Basisklasse ein.**
$BaseVirtual	**fügt 1 für virtuelle Basisklassen ein, sonst 0.**
$BaseAccessMode	**fügt 1 für public, 2 für protected und 3 für private ein.**
$BaseNo	**fügt die Nummer der Basisklasse ein (z.B. 4 für die an vierter Stelle deklarierte Basislasse).**
$BaseIsA(<Class>)	**liefert True, wenn die Basisklasse typkompatibel zu <Class ist.**

[1] Genauer gesagt: für jede Klasse, die in einer Datei deklariert wurde, die seit dem letzten Lauf von MopGen geändert wurde.

$BaseIsVirtual	liefert True, wenn die Basisklasse virtual geerbt wurde.
$BaseIsPublic	liefert True, wenn die Basisklasse public geerbt wurde.
$BaseIsProtected	liefert True, wenn die Basisklasse protected geerbt wurde.
$BaseIsPrivate	liefert True, wenn die Basisklasse private geerbt wurde.

8.2.4 Exemplar- und Klassenvariablenspezifische Anweisungen

Die folgenden Anweisungen sind nur innerhalb einer $ForEachField-Anweisung erlaubt und beziehen sich auf die Exemplar-/Klassenvariable, für die gerade Code erzeugt wird.

$FieldName	**fügt den Namen der Variablen ein**
$FieldDeclFile	**fügt den Namen der Datei ein, in der der Typ der Variable deklariert wurde (für #include-Anweisungen). Vorher muß mit $FieldIsA oder $Not $FieldIsBuiltin sichergestellt werden, daß die aktuelle Exemplarvariable einen non-built Typ hat.**
$FieldAccessMode	**fügt 1 für public, 2 für protected und 3 für private ein.**
$FieldConst	**fügt 1 für konstante Variablen ein, sonst 0.**
$FieldStatic	**fügt 1 für Klassenvariablen ein, 0 für Exemplarvariablen.**
$FieldArraySize	**fügt die Array-Größe der Variablen ein (1 für normale Variablen).**
$FieldNo	**fügt die Nummer der Variablen ein (z.B. 4 für die an vierte Stelle deklarierte Variable).**
$FieldBaseType	**fügt den Namen der Klasse der Variablen ein (für Built-In' nicht erlaubt).**
$FieldIsA(<Class>)	**liefert true, falls die Variable typkompatibel zu <Class> ist**
$FieldIsConst	**Dieser und die folgenden Ausdrücke liefern true, falls der T der Variablen die entsprechende Eigenschaft hat.**
$FieldIsStatic	
$FieldIsObject	**true, falls Variable kein Pointer, Array oder Reference ist.**
$FieldIsPointer	
$FieldIsDoublePointer	
$FieldIsReference	
$FieldIsArray	
$FieldIsComposite	**true, falls Variable mehr als einen Typ-Modifizierer hat (z. char**, Sample&*, int[10][20]).**
$FieldIsBuiltin	**true, falls Variable einen Built-In-Typ hat.**
$FieldIsCharType	
$FieldIsIntType	
$FieldIsFloatType	
$FieldIsLong	
$FieldIsSigned	
$FieldIsPublic	**Dieser und die folgenden Ausdrücke liefern true, falls die aktuelle Variable entsprechend deklariert wurde.**
$FieldIsProtected	
$FieldIsPrivate	

8.2.5 Methodenspezifische Anweisungen

Die folgenden Anweisungen sind nur innerhalb einer $ForEachMethod-Anweisung erlaubt.

$MethodName	**fügt den Namen der Methode ein.**
$MethodNo	**fügt die Nummer der Methode ein.**
$MethodIsVirtual	**Dieser und die folgenden Ausdrücke liefern true, falls die Methode entsprechend deklariert wurde.**

$MethodIsPublic	
$MethodIsProtected	
$MethodIsPrivate	

Die folgenden Anweisungen sind nur innerhalb einer $ForEachParam-Anweisung erlaubt.

$ParamName	fügt den Namen des Parameters ein.
$ParamDecl	fügt die Deklaration des Parameters ein.
$ParamTypeAsString	fügt eine Codierung des Parametertyps als String ein (z.B. IntPtr, ConstDouble, UnsignedCharPtr). Fügt für non-builtin-Typen Obj ein (z.B. ConstObjPtr, ObjPtrPtr).
$ParamNo	fügt die Nummer des Parameters ein.
$ParamIsConst	Dieser und die folgenden Ausdrücke liefern True, falls der Typ des aktuellen Parameters die entsprechende Eigenschaft hat. Weitere Erläuterungen s. im Abschnitt 8.2.4.
$ParamIsStatic	
$ParamIsPointer	
$ParamIsReference	
$ParamIsObject	
$ParamIsArray	
$ParamIsComposite	
$ParamIsBuiltin	
$ParamIsDoublePointer	
$ParamIsCharType	
$ParamIsIntType	
$ParamIsFloatType	
$ParamIsLong	
$ParamIsSigned	
$ParamIsLast	liefert True, falls der aktuelle Parameter der letzte Parameter der Methode ist.

8.2.6 Sonstige Anweisungen

$$	fügt nichts ein. Nützlich, wenn auf eine MOP-Anweisung direkt ein String folgen soll, wie z.B. in
	`$ClassName* $ClassName$$Creator()`
	`{`
	`  return new $ClassName;`
	`}`

$ClassFlagIsSet(<Name> **)**
$FieldFlagIsSet(<Name> **)**
$MethodFlagIsSet(<Name>**)**
	Innerhalb einer Klassendeklaration können mit Hilfe von Makros Flags gesetzt werden, um bestimmte Klassen, Exemplarvariablen und Methoden zu kennzeichnen:
	`SET_CLASS_FLAG( flag )`
	`SET_FIELD_FLAG( flag )`
	`RESET_FIELD_FLAG( flag )`
	`SET_METHOD_FLAG( flag )`
	`RESET_METHOD_FLAG( flag )`
	Diese Flags können mit den $...FlagIsSet-Ausdrücken abgefragt werden.
	Typische Anwendungsfälle sind die Kennzeichnung von Exemplarvariablen als persistent und die Kennzeichnung von Methoden als dynamisch aufrufbar (für Scripting).

$Error(<Text> **)**	Gibt den Text <Text> aus und bricht die Erzeugung für die aktuelle Klasse ab (wie die #`Error`-Anweisung des C-Preprozessors).
$Warning(<Text> **)**	wie $Error, aber die Verarbeitung der aktuellen Klasse wird nicht abgebrochen.

9 Die technische Realisierung von MopGen

9.1 Die Erzeugung der Meta-Information

Der für das MOP zuständige zusätzliche Code wird in drei Schritten erzeugt:

1. Die benötigte Information wird aus dem vorhandenen Quelltext extrahiert und in Hilfsdateien in leicht maschinenlesbarer Form abgelegt (CoiGen).
2. Die abgelegte Information und die MOP-Spezifikationsdatei werden eingelesen und zusätzlicher Quelltext erzeugt (MopGen).
3. Der zusätzliche Quelltext wird übersetzt (C++-Compiler).

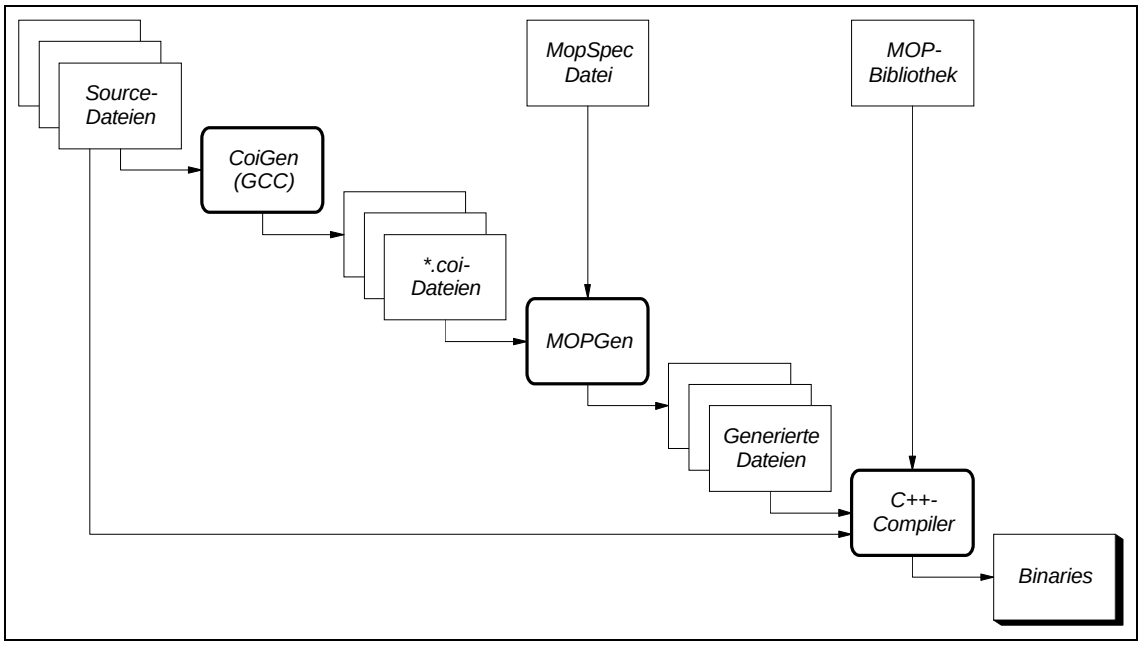

Diagramm 9.1 Der Übersetzungsablauf mit MopGen

Dieses Vorgehen hat mehrere Vorteile:

- Die extrahierte Information kann auch von anderen Programmen verwendet werden (z.B. Klassenbrowser).
- Da nicht zusätzlicher Maschinen-Code, sondern zusätzlicher Quelltext erzeugt wird, kann jeder beliebige Compiler zur Übersetzung verwendet werden.

Dem steht als Nachteil gegenüber, daß die Performance der Erzeugung nicht optimal ist. Da ein neuer MOP-Erzeugungslauf jedoch nur notwendig ist, wenn sich eine Typdefinition geändert hat, ist dieser Nachteil nicht so schwerwiegend.

9.1.1 Die Extraktion der Klasseninformation

Zur Extraktion der Klasseninformation wird der C++-Compiler GCC verwendet. Dieser Compiler wird von der Free Software Foundation kostenlos zur Verfügung gestellt. Der Quelltext ist ebenfalls verfügbar, sodaß eine Modifikation durchgeführt werden konnte.
Zunächst wird für jeden Typ eine Datei erzeugt, die die nötigen Typinformationen enthält. Diese Informationen werden dem attributierten Parse-Tree entnommen. Jeder Typ erhält eine projektglobal eindeutige Id. Da der Compiler die einzelnen Quelltextfiles in getrennten Compilerläufen übersetzt, werden die bisher vergebenen Id's am Ende jeder Übersetzung in einer Hilfsdatei gespeichert und zu Beginn einer neuen Übersetzung wieder eingelesen.
Eine erzeugte Klassenbeschreibung hat folgende Form (am Beispiel der Klasse `tRectangle` aus Abschnitt 8.1):

```
Class tRectangle
  File: rectangl.cxx
  Attributes: has_virtual_baseclasses has_virtual_functions

  Baseclass tGraphObject
    Attributes: public virtual

  Field x
    Type: double
    Attributes: private
    Flags: Persistent

  ...   // weitere Felder

  Field buffer
    Type: * tBitmap
    Attributes: private

  Method GetAsBitmap
    Attributes: public virtual const
    Return Type: const * tBitmap
    Parameter this
      Type: * Child3

  ...  // weitere Methoden
```

Typen werden von CoiGen kanonisiert, d.h. die Deklaration

```
typedef class Example* ExamplePtr;

class Example
{
    ExamplePtr next;
    ...
}
```

führt zur Ausgabe

```
Field next
    Type: * Example
    Attributes: private
```

In der CoiGen-Ausgabe tauchen also mit `typedef` definierte Typen nicht mehr auf.

9.1.2 Die Erzeugung des Quelltextes für das Laufzeit-MOP

MopGen liest zur Erzeugung des MOPs die Klassenbeschreibungen und eine Datei mit dem MopSpec-Quelltext ein. Aus jeder Klassenbeschreibung wird ein Klassenmetaobjekt generiert. Auf diese Objekte wird bei der Auswertung des MopSpec-Quelltexts zurückgegriffen.
Der MopSpec-Code wird geparst und daraus ein Parse-Tree erstellt.
Anhand einer per Makefile erstellten Hilfsdatei kann MopGen erkennen, welche Dateien sich seit dem letzten Aufruf geändert haben. Für die in diesen Dateien deklarierten Klassen wird der erstellte Parse-Tree durchlaufen und der gewünschte MOP-Code erzeugt.

9.2 Einschränkungen

MopGen unterliegt zur Zeit noch einigen Einschränkungen. Diese Einschränkungen sind allerdings nicht prinzipbedingt, sondern hauptsächlich deshalb vorhanden, weil der Implementierungsaufwand sonst zu groß geworden wäre. Die Einschränkungen sind folgende:

- CoiGen liest nur die Header-Files ein und wertet daher nur die darin enthaltenen Typ-Deklarationen aus. Technisch gesehen wäre es möglich, auch die Implementations-Files auszuwerten; das hätte jedoch zur Folge, daß bei jeder Änderung der Implementation das MOP neu erzeugt werden müßte. Daher scheint es sinnvoller, Typ-Deklarationen nur in Header-Files zu erlauben.
- Templates können nicht behandelt werden.
- Zur Zeit kann nur für Klassen Meta-Information erzeugt werden., nicht für Aufzählungstypen, `union`s, usw.

10 Ein Beispiel-MOP mit MopGen

In diesem Kapitel ist das Beispiel-MOP beschrieben, das zusammen mit MopGen entwickelt wurde. Es soll an dieser Stelle noch einmal darauf hingewiesen werden, daß dieses MOP nicht an jeder Stelle vorbildlich ist. Z.B. könnte und sollte erheblich mehr Funktionalität in eigene Klassen-Metaobjekte verlagert werden. Der Hauptzweck dieses Beispiel-MOPs ist es, zu zeigen, wie ein praxistaugliches MOP mit Hilfe von MopGen und MopSpec realisiert werden kann. Der verwendete MopSpec-Code kann auch als Ausgangspunkt für die Entwicklung eines eigenen MOPs verwendet werden. Er ist in Anhang A beschrieben.

Um eine Klasse in den „Genuß“ der MOP-Funktionalität kommen zu lassen, muß man sie von tHasMop erben lassen. Sollen auch Methoden für Object-I/O erzeugt werden, muß zusätzlich die Aspektklasse tStorable geerbt werden. Die Deklaration der zusätzlichen Methoden und Klassenvariablen werden durch die Makros `MetaDef( classname )`, `StoreDef( classname )` und `GCDef( classname )` in die Klassendeklaration eingefügt. Z.T. sind sie nur in tHasMop deklariert, weil sie nicht überschrieben werden müssen. Folgende Beispielklasse veranschaulicht das Vorgehen:

```
class tSample : public virtual tHasMop,
                public virtual tStorable,   // nur für Object-I/O
                public virtual tCollectable // nur, falls GC verwendet wird
{
    MetaDef( tSample );
    StoreDef( tSample );
    GCDef( tSample );
  public:
    ... // der Rest der Deklaration
};
```

10.1 RTTI

Die RTTI-Schnittstelle besteht aus folgenden Methoden:

```
static  const  class tTypeInfo& GetTypeInfoStatic();
virtual const  class tTypeInfo& GetTypeInfo() const;
virtual unsigned int IsA( const class tTypeInfo& class ) const;
virtual unsigned int IsKindOf( const class tTypeInfo& class ) const;
virtual const char* GetClassName() const;
```

Für jede Klasse wird ein tTypeInfo-Metaobjekt erzeugt. Mit `GetTypeInfo()` bzw. `GetTypeInfoStatic()` erhält man eine Referenz auf dieses Metaobjekt. Die `tTypeInfo`-Objekte sind entsprechend dem Klassenbaum miteinander verbunden.
`IsA( tTypeinfo& class )` liefert true, wenn das Objekt ein Exemplar der übergebenen Klasse ist. Mit der Methode `IsKindOf( tTypeinfo& class )` kann man testen, ob ein Objekt typkompatibel zu der übergebenen Klasse ist.

10.2 Downcasting

Jede Klasse erhält zwei statische Methoden Cast() (jeweils für konstante und nicht-konstante Objekte). Mit Hilfe dieser Methode läßt sich ein überprüfter Cast durchführen.

```
static tSample*       Cast( tHasMop*);
static const tSample* Cast( const tHasMop* );
```

Der Cast liefert eine Referenz vom gewünschten Typ, falls das übergebene Objekt typkompatibel (`IsKindOf()`) zum gewünschten Typ ist; andernfalls wird der Null-Pointer zurückgegeben. Der Cast funktioniert im Gegensatz zum C++-Standardcast auch bei Mehrfachvererbung mit Verwendung virtueller Basisklassen. Dieselbe Art des Casts soll zukünftig Bestandteil von C++ werden (s. 6.3). Die Cast-Methode wird folgendermaßen benutzt:

```
...
```

```
tHasMop* pVar = Container[i];
tFaxDoc* pFax = tFaxDoc::Cast( pVar );
if ( pFax )
{
     // es ist ein Fax
     pFax->Send( TelNr );
     ...
}
else
{
     // kein Fax
     ...
}
...
```

10.3 Späte Erzeugung

Späte Erzeugung ermöglicht es, durch Angabe von Eigenschaften ein Objekt zu erzeugen, dessen genauen Typ man nicht kennt [Riehle95]. Dazu ist es nötig, daß eine Create-Anforderung im Vererbungsbaum nach unten gereicht werden kann. Jede Klasse erhält eine statische Methode `CreateLateStatic()` und ein virtuelle Methode `CreateLate()`, die als Parameter eine Anforderungsbeschreibung erhalten:

```
virtual tHasMop* CreateLate( const tClause& ) const;
static tSample* CreateLateStatic( const tClause& );
```

`CreateLateStatic()` ruft die `CreateLate()`-Methode eines prototypischen Exemplars ihrer Klasse auf. Diese Methode ruft zunächst die `CreateLate()`-Methoden der Prototypen ihrer Unterklassen auf. Falls eine der Prototypen die Anforderungen erfüllen konnte und ein entsprechendes Objekt erzeugt hat, wird es an den Aufrufer zurückgegeben. Andernfalls prüft die `CreateLate()`-Methode, ob sie nicht selbst die Anforderungen erfüllen kann. Falls ja, liefert sie das gewünschte Objekt zurück; falls nein, meldet sie den Fehlschlag nach oben und die Suche wird in dem nächsten Zweig des Vererbungsbaumes fortgesetzt.
Das Erzeugen eines neuen Exemplars einer Klasse, deren Name als String übergeben wird, ist ein Spezialfall der späten Erzeugung; diese Funktionalität wird im Beispiel-MOP zur Verfügung gestellt, ohne daß der Entwickler selbst Anforderungsbeschreibungen schreiben muß.
Zur genauen Beschreibung der Implementation s. [Riehle95].

10.4 Object-I/O

Als Anwendungsbeispiel wurden Methoden generiert, mit deren Hilfe Objekte als Text auf C++-Streams ausgegeben werden können. Der Text kann einerseits als Debugging-Information verwendet werden, da auch die Namen der Exemplarvariablen ausgegeben werden und das Ausgabeformat „menschenlesbar“ gewählt wurde. Der ausgegebene Text kann allerdings auch wieder als Eingabe verwendet werden, aufgrund derer die Objekte neu erzeugt werden können.
Aufgrund der überflüssigen Informationen und des gewählten Text-Formats eignet sich diese Art des Object-I/O nicht, um größere Datenmengen häufig zu speichern und zu lesen.
Während die in Kap. 6 vorgestellten Object-I/O-Konzepte Referenzen auf eingebettete Objekte nicht korrekt behandeln, kann der hier vorgestellte Algorithmus auch damit umgehen. Dies wird durch eine zweistufige Vorgehensweise möglich: Im ersten Schritt werden die Adressen aller Objekte in eine Liste eingetragen, auf die von den abzuspeichernden Objekten aus verwiesen wird. Falls ein Objekt eingebettete Objekte enthält, werden auch die Adressen dieser Objekte in die Liste eingetragen und mit einem „embedded“-Flag gekennzeichnet. Im zweiten Schritt werden alle in der Liste eingetragenen Objekte, die nicht als „embedded“ gekennzeichnet sind, auf den Stream geschrieben.
Dieses Vorgehen ist mit der Klasse `tStore` modelliert:

```
class tStore
{
    int ObjCount;
    tDynArray<tStorObj> ObjTbl;

    public:
    void Insert( class tStorable& obj );
    void SaveOn( class ostream& OutStr );
    void ReadFrom( class istream& InStr );
    ...
};
```

Jedes Objekt, das abgespeichert werden soll, muß mit `Insert()` in die Objektliste eingetragen werden.
`SaveOn()` vervollständigt die Objektliste, gibt zunächst diese Liste aus und schreibt dann die Objekte auf den übergebenen Stream. Die Verweise werden dabei als Index in der Objektliste abgespeichert.

```
void tStore::SaveOn( ostream& os )
{
    Complete();             // Objektliste anhand der Referenzen auf
                            // andere Objekte vervollständigen

    WriteObjList( os );     // Liste aller Objekte (mit Index, Klassenname
                            // und Embedded-Flag) ausgeben

    for( unsigned long i = 1; i <= ObjCount; i++ )
    {   // Objekte ausgeben, wenn sie nicht eingebettet sind
        if ( ! ObjTbl[i].IsEmbedded )
        {
            os << "Object " << i << ":" << endl;
            ObjTbl[i].obj->WriteOnStore( *this );
            os << endl;
        }
    }
}

void tStore::Complete()
{
    for( unsigned long i = 1; i <= ObjCount; i++ )
    {   // Alle Verweise (auch auf eingebettete Objekte) in Liste eintragen
        ObjTbl[i].obj->PutReferences( *this );
    }
}
```

Die Methoden `WriteOnStore` und `PutReferences` werden von MopGen generiert (s. 12.7) und haben folgende Deklaration:

```
virtual void WriteOnStore( class tStore& store );
virtual void ReadFromStore( class tStore& store );
```

Das Einlesen funktioniert folgendermaßen: Zunächst wird die Objektliste eingelesen und jedes nicht-eingebettete Objekt mit Hilfe der späten Erzeugung (s. 10.3) erzeugt. Als nächstes werden alle Objekte eingelesen. Zum Schluß werden die Index-Nummern der Objekte durch die eigentlichen Adressen ersetzt.

```
void tStore::ReadFrom( istream& is )
{
    ReadObjList( is );
    for( unsigned long i = 1; i <= ObjCount; i++ )
    {
        if ( ! ObjTbl[i].IsEmbedded )
        {
            ObjTbl[ObjNo].obj->ReadFromStore( *this );
        }
    }
```

```
        for( i = 1; i <= ObjCount; i++ )
        {
            if ( ! ObjTbl[i].IsEmbedded )
            {
                ObjTbl[i].obj->ReadPtrFromStore( *this );
            }
        }
    }

    void tStore::ReadObjList( istream& is )
    {
        tString ClassName;
        unsigned long ObjNo;

        is >> ObjCount;
        for( unsigned long i = 1; i <= ObjCount; i++ )
        {
            is >> ObjNo >> ClassName >> ObjTbl[ObjNo].IsEmbedded;
            if ( ObjTbl[ObjNo].IsEmbedded )
            {
                ObjTbl[ObjNo].obj = NULL;
            }
            else
            {
                tTypeClause tc( ClassName );
                ObjTbl[ObjNo].obj = tStorable::CreateLateStatic( tc );
            }
        }
    }
```

10.5 Garbage Collection

Zur Durchführung der Garbage Collection muß ein Collector die Menge der Objekte bestimmen, auf die noch Referenzen existieren. Das Beispiel-MOP erzeugt die Methode `CallbackPointers()`, die dem Collector die alle Referenzen auf andere Objekte liefert, die in einem Objekt enthalten sind:

```
virtual void CallbackPointers();
```

10.6 Dynamischer Methodenaufruf

Das Beispiel-MOP stellt die Methode `CallMethod()` zur Verfügung, die es erlaubt, andere Methoden durch Übergabe des Namens aufzurufen:

```
virtual void CallMethod( tString& MethodName,
                         tList<GenericArg>& Args,
                         GenericArg& Ret );
```

Die Argumente müssen als `GenericArg`-Array übergeben werden, der Return-Wert wird als `GenericArg` zurückgeliefert. Overloading by arguments ist nicht möglich, d.h. die Methode muß eindeutig durch ihren Namen identifizierbar sein.

11 Ausblick und Zusammenfassung

11.1 Ausblick

MopGen ist zwar ein funktionsfähiges System, das auch schon erfolgreich angewendet wurde, nämlich auf sich selbst: Für alle Klassen von MopGen wird das Beispiel-MOP generiert; RTTI, Downcasting und späte Erzeugung werden an mehreren Stellen innerhalb von MopGen verwendet. Um „produktionsreif“ zu werden, müßte jedoch noch an einigen Punkten gearbeitet werden.
Die höchste Priorität hätte dabei die Syntax von MopSpec. Das einzig Gute, was man von ihr behaupten kann, ist, daß es extrem einfach ist, einen Parser für MopSpec zu schreiben; aus diesem Grund hat sie auch ihre jetzige Form. Das führt allerdings zu einer überflüssigen Aufblähung des Befehlssatzes und zu einer gewissen Unübersichtlichkeit. Ein großer Schritt in Richtung einer brauchbaren Syntax wäre die Einführung von Variablen: Dann könnte man z.B. anstatt

```
$ForEachField {
    $If ( FieldIsProtected ) {
        $FieldName
...
```

das klarere

```
$ForEachField( f ) {
    $If ( f.IsProtected() ) {
        f.Name()
...
```

schreiben.
Man müßte auch eine Form finden, die sich mehr an die Gepflogenheiten von C++ anlehnt (vielleicht eine `metatemplate-Anweisung`?).
Außerdem müßten die in Abschnitt 9.2 erwähnten Einschränkungen beseitigt werden. Dabei steht an erster Stelle die Möglichkeit, auch für Klassen-Templates Code zu generieren.

11.2 Zusammenfassung

Meta-Informationen sind für größere Anwendungen unabdingbar, wie die vielen Entwicklungen für C++ zeigen (Kap. 6). Die MOPs der untersuchten schwach getypten Interpretersprachen (Smalltalk, CLOS) können vom Entwickler modifiziert und erweitert werden, ohne die schon bestehende MOP-Funktionalität zu beeinflussen. Die MOPs der stark getypten Compilersprachen (Eiffel, C++) haben diese Eigenschaft leider nicht. Ihre Funktionalität ist festgelegt und nicht vom Entwickler modifizierbar. Dies gilt auch für die in Kap. 6 vorgestellten Precompiler. Die Makro-basierten MOPs für C++ sind zwar vom Entwickler modifizierbar; die Tatsache, daß so viele MOPs für C++ mit einem Precompiler realisiert wurden, ist jedoch ein Indiz dafür, daß Makro-basierte Lösungen an ihre Grenzen stoßen, wenn es um die Implementierung komplexer MOP-Funktionalität geht (z.B. Methodenumlenkung). Wer allerdings nur ein relativ kleines MOP benötigt, ist beim heutigen Stand der Technik mit einer Makro-basierten Implemtierung am besten bedient.
MopGen ist ein Versuch, die Vorteile der Modifizierbarkeit und insbesondere Erweiterbarkeit eines MOP mit den Vorteilen der Generierung des MOP durch einen Precompiler zu verbinden. Der Gewinn liegt darin, daß mit MopGen ein optimal an die eigene Anwendung angepaßtes MOP verwendet werden kann.
Dazu wurde eine eigene Sprache, MopSpec, eingeführt. Dies hat natürlich den Nachteil, daß der Umgang mit dieser Sprache erst einmal gelernt werden muß. Die Komplexität von MopSpec ist allerdings nicht besonders groß, eher vergleichbar mit der des C-Preprozessors als mit der einer „richtigen“ Programmierprache. Der anfänglich höhere Aufwand macht sich daher wahrscheinlich schon nach kurzer Zeit bezahlt. Dies gilt insbesondere dann, wenn die Alternative die Entwicklung des 143. eigenen Precompilers ist.

Anhang

12 Anhang A: Der MopSpec-Code des Beispiel-MOP

12.1 Die Deklarationen

Folgende Methoden und Exemplarvariablen werden mittels der Makros `MetaDef()` und `StoreDef()` deklariert (am Beispiel der Klasse `tSample`):

```
class tSample : public virtual tHasMop, public virtual tStorable
{
    /* RTTI */
    static  const tTypeInfo& GetTypeInfoStatic();
    virtual const tTypeInfo& GetTypeInfo() const;

    /* Casting */
  public:
    static name*       Cast( tHasMop*);
    static const name* Cast( const tHasMop* );
  protected:
    typedef tSample* (*tSampleCastFkt)( tHasMop* );
    static  tDumbList< tSampleCastFkt > &GetCastFktList() ;
    static unsigned int RegisterCastHelper( tSampleCastFkt cast );
    virtual void*      GetAdr() const;

    /* Late Creation */
  public:
    virtual tHasMop* CreateLate( const tClause& ) const;
    static name* CreateLateStatic( const tClause& );
  protected:
    static  const tHasMop* ThisPrototype;
    static const tHasMop* GetPrototype();
    virtual tDumbList< const tHasMop* > & GetSubclassList() const;

    /* Object-I/O */
  public:
    friend class tStore;
    virtual void WriteOnStore( class tStore& store );
    virtual void ReadFromStore( class tStore& store );
    virtual void ReadPtrFromStore( class tStore& store );
    virtual void PutReferences( class tStore& store );
    virtual void ResetStoreFlags();
    private:
    bool IsWrittenOrRead;

    /* Garbage Collection */
    ...
};
```

12.2 Die #include-Anweisungen

In diesem und den folgenden Abschnitten wird Stück für Stück der MopSpec-Code für das Beispiel-MOP vorgestellt. Dieser Quelltext wird immer wieder von Kommentaren unterbrochen, ist aber als eine Datei zu verstehen.
Das MOP wird nur für Klassen erzeugt, die von `tHasMop` erben:

```
$If ( $ClassIsA( tHasMop ) )
{
```

Als erstes werden die #`include`-Anweisungen für die benötigten Header-Dateien generiert:

```
#include "$ClassDeclFile"   // Header-File der bearbeiteten Klasse
#include "typeinfo.hxx"     // tTypinfo-Klasse
#include "createcl.hxx"     // Klausel-Klasse tClause für späte Erzeugung

#include <iostream.h>
```

12.3 Die Initialisierung

Für jede Klasse wird ein prototypisches Objekt erzeugt. Diese Prototypen werden entsprechend dem Klassenbaum verkettet, d.h. jeder Prototyp meldet sich bei den Prototypen seiner Oberklassen an. Eine bessere Lösung ist die Verkettung von Klassen-Metaobjekten.
Die Deklaration von `ThisPrototype` sorgt dafür, daß der Prototyp jeder Klasse vor `main()` erzeugt wird.

```
const tHasMop* $ClassName::ThisPrototype = $ClassName::GetPrototype();
```

`GetPrototype()` erzeugt das prototypische Objekt. Durch Aufruf der `Register()`-Methode wird der Prototyp bei den Prototypen der Oberklassen angemeldet. Zur Zeit existiert für die `Cast`-Funktionen eine eigene Datenstruktur: sog. CastHelper-Funktionen werden in Listen eingetragen (s. 12.5).
Der erzeugte oder schon vorhandene Prototyp wird zurückgeliefert.

```
const tHasMop* $ClassName::GetPrototype()
{
    static tHasMop* Proto = 0;

    if ( Proto == 0 )
    {
        Proto = new $ClassName();
        $ForEachBase
        {
            ((tHasMop *)$BaseName::GetPrototype())->Register( Proto );
            $BaseName::RegisterCastHelper(
                        CastHelper_$ClassName$$_to_$BaseName );
        }
    }

    return Proto;
}
```

Damit sichergestellt ist, daß die `SubclassList` vor der ersten Benutzung erzeugt wird, wird sie als statische Variable einer Methode deklariert.

```
tDumbList<const tHasMop*>& $ClassName::GetSubclassList() const
{
    static tDumbList<const tHasMop*> SubclassList;

    return SubclassList;
}
```

12.4 RTTI

Die statische `GetTypeInfo()`-Methode liefert das `tTypeInfo`-Objekt des Prototypen zurück.

```
const tTypeInfo& $ClassName::GetTypeInfoStatic()
{
    return $ClassName::GetPrototype()->GetTypeInfo();
}
```

Das `tTypeInfo`-Objekt ist eine statische Variable der `GetTypeInfo()`-Methode. Es enthält den Namen der Klasse und eine Liste der `tTypeInfo`-Objekte der Basisklassen.

```
const tTypeInfo& $ClassName::GetTypeInfo() const
{
    static tTypeInfo::tBaseInfo BaseInfo[] = {
  $ForEachBase
  {
    {&$BaseName::GetTypeInfo(), $BaseAccessMode, $BaseVirtual} ,
  }
    {0, 0, 0} };
```

```
    static tDumbList<tTypeInfo::tBaseInfo>
            Bases( BaseInfo,
               sizeof(BaseInfo)/sizeof(tTypeInfo::tBaseInfo)-1 );

    static char Name[] = "$ClassName" ;

    static tTypeInfo TypeInfo( Name,
                               &Bases );

    return TypeInfo;
}
```

12.5 Downcasting

Die Implementation des Downcasts ist so undurchsichtig, daß es ein hoffnungsloses Unterfangen wäre, sie an dieser Stelle vollständig zu erklären. Sie bedürfte dringend eines Redesigns. Aber es funktioniert!

```
$ForEachBase
{
    $BaseName* CastHelper_$ClassName$$_to_$BaseName( tHasMop* o)
    {
        $BaseName* ret = $ClassName::Cast( o );
        return ret;
    }
}

tDumbList<$ClassName::$ClassName$$CastFkt>& $ClassName::GetCastFktList()
{
    static tDumbList<$ClassName$$CastFkt> CastFktList;

    return CastFktList;
}

unsigned int $ClassName::RegisterCastHelper( $ClassName$$CastFkt cast )
{
    return GetCastFktList().Insert( cast );
}
```

`GetAdr()` liefert die Adresse des Objekts als `void*` zurück. Diese Adresse wird in der `Cast()`-Routine wieder in den ursprünglichen Typ des Objekts zurückgewandelt. Der C++-Standard garantiert, daß dieses Hin- und Zurückcasten den korrekten `this`-Pointer liefert.

```
void* $ClassName::GetAdr() const
{
    return (void*) this;
}

$ClassName* $ClassName::Cast( tHasMop* p )
{
    if( p == 0)
        return 0;
    if(  p->GetTypeInfo() == $ClassName::GetTypeInfoStatic() )
    {
        return ($ClassName*) p->GetAdr(); //hier wird hin- und hergecastet
    }
    else if ( p->IsKindOf( $ClassName::GetTypeInfoStatic() ) )
    {
        tDumbIter< $ClassName$$CastFkt > iter(
                                    $ClassName::GetCastFktList() );

        while( iter.IsValid() )            // for each castfkt
        {
            $ClassName$$CastFkt cf = iter.Next();
            $ClassName* ret = cf( p );
```

```
            if ( ret != 0 )
            {
                return ret;
            }
        }
        return 0;
    }
    else
    {
        return 0;
    }
}

// die const-Variante
const $ClassName* $ClassName::Cast( const tHasMop* p )
{
    return $ClassName::Cast( (tHasMop*) p );
}
```

12.6 Späte Erzeugung

Zur Implementierung der späten Erzeugung s. [Riehle95]

```
$ClassName* $ClassName::CreateLateStatic( const tClause& clause )
{
     return $ClassName::Cast(
            $ClassName::GetPrototype()->CreateLate( clause ) );
}

tHasMop* $ClassName::CreateLate( const tClause& clause ) const
{
    tHasMop* ret = 0;

    if (   IsFittingClause( clause )
        || clause.IsKindOf( tTypeClause::GetTypeInfoStatic() )
       )
    {
        tDumbIter<const tHasMop*> iter( GetSubclassList() );
        while( iter.IsValid() )
        {
            ret = iter.Next()->CreateLate( clause );
            if ( ret )
            {
                return ret;
            }
        }

        if ( IsValidClause( clause ) )
            ret = new $ClassName();
        else if ( clause.IsKindOf( tTypeClause::GetTypeInfoStatic() ) )
        {
             const tTypeClause* tc = tTypeClause::Cast( &clause );
             if ( GetTypeInfo().IsA( tc->GetTypeName() ) )
                ret = new $ClassName();
        }
    }

    return ret;
}
```

12.7 Object-I/O

Die Implementierung des Object-I/O ist in Abschnitt 10.4 genau erklärt

```
$If ( $ClassIsA( tStorable ) ) {
#include "store.hxx"
```

```
void $ClassName::ResetStoreFlags()
{
    $ForEachBase
    {
      $If ( $BaseIsA( tStorable ) )
      {
        $BaseName::ResetStoreFlags();
      }
    }
    IsWrittenOrRead = 0;
}

void $ClassName::WriteOnStore( tStore& store )
{
    if ( IsWrittenOrRead )
        return;
    IsWrittenOrRead = 1;

    $ForEachBase
    {
      $If ( $BaseIsA( tStorable ) )
      {
        $BaseName::WriteOnStore( store );
      }
    }

    $ForEachField
    {
      $If ( $IsPersistent )
      {
        $If ( $IsBuiltin )
        {
          $If ( $IsObject )
          {
              store.WriteBuiltIn( "$FieldName", &$FieldName );
          }
          $Else
          {
              $Warning( pointer or references to built-in types
                        can't be stored )
              // $FieldName is unstorable
          }
        }
        $Else {
          $If ( $FieldIsA( tStorable ) )
          {
              $If ( $IsObject )
              {
                  store.WriteObject( "$FieldName", &$FieldName );
              }
              $If ( $IsPointer )
              {
                  store.WritePointer( "$FieldName", $FieldName );
              }
              $If ( $IsComposite $Or $IsArray $Or $IsReference )
              {
                  $Warning( ComplexTypes,Arrays and References
                            can't be stored. )
                  // $FieldName is unstorable
              }
          }
```

```
            $Else
            {
                $Warning(Objects must inherit from tStorable to be storable)
                // $FieldName is unstorable
            }
          }
        }
      }
    }

    void $ClassName::ReadFromStore( tStore& store )
    {
        if ( IsWrittenOrRead )
            return;
        IsWrittenOrRead = 1;

        $ForEachBase
        {
          $If ( $BaseIsA( tStorable ) )
          {
            $BaseName::ReadFromStore( store );
          }
        }

        $ForEachField
        {
          $If ( $IsPersistent )
          {
            $If ( $IsBuiltin $And $IsObject )
            {
                  store.ReadBuiltIn( "$FieldName", &$FieldName );
            }
            $If ( $FieldIsA( tStorable ) )
            {
                $If ( $IsObject )
                {
                    store.ReadObject( "$FieldName", &$FieldName );
                }
                $If ( $IsPointer )
                {
                    $FieldName = ($FieldType)store.ReadPointer("$FieldName");
                }
            }
          }
        }
    }

    void $ClassName::ReadPtrFromStore( tStore& store )
    {
        if ( IsWrittenOrRead )
            return;
        IsWrittenOrRead = 1;

        $ForEachBase
        {
          $If ( $BaseIsA( tStorable ) )
          {
            $BaseName::ReadPtrFromStore( store );
          }
        }
```

```
    $ForEachField
    {
      $If ( $IsPersistent )
      {
        $If ( $FieldIsA( tStorable ) )
        {
            $If ( $IsPointer )
            {
                $FieldName = $FieldBaseType::Cast(
                    (tHasMop*)store.InitPointer( $FieldName ) );
            }
            $If ( $IsObject )
            {
                $FieldName.ReadPtrFromStore( store );
            }
        }
      }
    }
}

void $ClassName::PutReferences( tStore& store )
{
    if ( IsWrittenOrRead )
        return;
    IsWrittenOrRead = 1;

    $ForEachBase
    {
      $If ( $BaseIsA( tStorable ) )
      {
        $BaseName::PutReferences( store );
      }
    }

    $ForEachField
    {
      $If ( $IsPersistent )
      {
        $If ( $FieldIsA( tStorable ) )
        {
            $If ( $IsObject )
            {
                store.PutObjRef( &$FieldName );
            }
            $If ( $IsPointer )
            {
                store.PutObjPtr( $FieldName );
            }
            $If ( $IsComposite $Or $IsArray $Or $IsReference )
            {
                $Warning( ComplexTypes,Arrays and References can't
                          be stored. )
                // $FieldName is unstorable
            }
        }
      }
    }
}
} // End of ClassIsA( tStorable )
```

12.8 Garbage Collection

```
$If ( $ClassIsA( tCollectable ) )
{
#include "gc.hxx"
#include "$ClassDeclFile"

$ForEachField
{
    $If ( $Not $IsBuiltin )
    {
#include "$FieldDeclFile"
    }
}

void $ClassName::ResetGCFlags()
{
    $ForEachBase
    {
        $If ( $BaseIsA( tCollectable) )
        {
            $BaseName::ResetGCFlags();
        }
    }
    GCFlag = 0;
}

void $ClassName::CallbackPtr()
{
    if ( GCFlag )
        return;
    GCFlag = 1;

    $ForEachBase
    {
        $If ( $BaseIsA( tCollectable ) )
        {
            $BaseName::CallbackPtr();
        }
    }

    $ForEachField
    {
        $If ( $IsPointer $And $FieldIsA( tCollectable ) )
        {
            vUsedPointer( $FieldName );
    }
        $If ( ( $IsObject $Or $IsReference )
              $And $FieldIsA ( tCollectable ) )
        {
            $FieldName.CallbackPointers();
        }
        $If ( $IsComposite $Or $IsArray )
        {
            $Warning( ComplexTypes And Arrays can't be collected.
                      Use tPointer or tArray instead.
                    )
            // Field $FieldName is uncollectable
        }
    }
}
}  // End of ClassIsA( tCollectable )
```

12.9 Dynamischer Methodenaufruf[1]

```
bool CallMethod(  const tString& MethodName,
                  tArray<GenericArg>& Args,
                  GenericArg& Ret )
{
    $ForEachMethod{
        $If( $MethodFlagIsSet( Dispatchable ) )
        {
            if ( MethodName == "$MethodName" )
            {
                i = 0;
                Ret = $MethodName(
                ForEachParam
                {
                    $If( $ParamIsBuiltIn )
                    {
                        Args[i++].
                        Get$ParamBaseType$ParamTypeSign$ParamTypeMod$$()
                        // for unsigned long * this expands to
                        // Args[i++].GetLongUnsignedPtr()
                    }
                    $Else $If ( $ParamIsA( Point ) )
                    {
                        ParamArray[i++].GetPoint$ParamTypeMod$$()
                    }
                    $Else $If ( $ParamIsA( Object ) )
                    {
                        ParamArray[i++].GetObject$ParamTypeMod$$()
                    }
                    $Else
                    {
                        $Error( This type is not allowed as parameter for
                                    dispatchable methods )
                    }
                    $If( $Not $IsLastParam )
                    { , }
                    $Else
                    { ); }
                }
                return true;
            }
        }
    }

    $ForEachBase
    {
        if ( $BaseName::CallMethod( MethodName, Args, Ret ) )
            return true;
    }
    return false;
}
} // End of ClassIsA( tHasMop )
```

[1] Das Method Dispatching befindet sich noch im Experimentierstadium.

13 Anhang B: MopSpec-Code für die C++-MOPs aus Kap. 6

Der hier vorgestellte Quelltext ist kein ausgetesteter „ready-to-run-code“. Solchen Code hier vorzustellen ist schon allein deswegen nicht möglich, weil der Quelltext der untersuchten MOPs zum größten Teil nicht veröffentlicht ist. Es läßt sich aber mit den hier vorgelegten Entwürfen zeigen, daß die Quelltext-Generatoren der untersuchten MOPs durch MopGen ersetzbar wären bzw. daß MopGen (bis auf kleine Einschränkungen, s. die einzelnen Abschnitte) dieselben Möglichkeiten bietet wie die Vereinigungsmenge aller untersuchten MOP-Generatoren.

13.1 Standard-C++-RTTI

Eine vollständige Nachbildung des RTTI-Standards mit MopGen ist nicht möglich, da keine neuen C++-Operatoren definiert werden können. Eine näherungsweise Implementation sähe so aus:

```
// Im Header-File:

class Sample : public RTTIClass
{
    public:
    static const type_info& GetClassInfo();
    virtual const type_info& GetTypeInfo(){ return GetClassInfo(); }
    static Sample* Cast( RTTIClass* p );
    private:
    // Verschiedene Cast-Hilfsfunktionen, s. 12.5
    ...
};

template <class T>
T* dynamic_cast( RTTIClass* p )
{
    return T::Cast( p );
}

const type_info& typeid( RTTIClass* p )
{
    return p->GetTypeInfo();
}


// MopSpec-Code
$If ( $ClassIsA( RTTIClass ) )
{
    // type_info-Objekt generieren
    const type_info& GetClassInfo()
    {
        static type_info ThisTypeInfo = type_info( $ClassName );
        return ThisTypeInfo;
    }

    $ClassName* Cast( RTTIClass* p )
    {
        ... // Die Spezifikation für das Downcastings findet sich in 12.5
    }
    ...
}
```

13.2 Metaflex

Da [JohPal93] leider nicht genügend Informationen über die Implementierung ihres MOP enthält, kann an dieser Stelle auch keine Spezifikation mit MopSpec angegeben werden. Die Spezifikation einer eigenen Implementierung mit derselben Funktionalität wie Metaflex wird in Abschnitt 12.9 beschrieben.

13.3 Das ET++-MOP

Die Methoden, die mittels eines einfachen Makros generiert werden können (z.B. `IsA()`), werden hier nicht berücksichtigt, da ihre Generierung mit MopGen trivial ist. Interessant sind dagegen diejenigen Methoden, die entweder komplett vom Entwickler selbst geschrieben werden müssen (`PrintOn`, `ReadFrom`) oder halbautomatisch erzeugt werden (`Members`):

```
$If( $ClassIsA( Object ) )
{
    ostream& $ClassName::PrintOn( ostream &s )
    {
        $ForEachBase
        {
            $BaseName::PrintOn( s );
        }

        $ForEachField
        {
            $If ( $FieldFlagSet( Persistent ) )
            {
                $If ( $FieldIsPointer $And $FieldIsCharType )
                {
                    PrintString( s, $FieldName );
                }
                $Else
                {
                    s << $FieldName << " ";
                }
            }
        }
        return s;
    }
}
```

In der Klassendefinition müssen die persistenten Exemplarvariablen markiert werden:

```
class ConnectionShape : public Shape
{
    SetMopGenFlag( Persistent );
    enum{ red, blue, green } color;
    Shape* startShape;
    Shape* endShape;
    char* description;
    ResetMopGenFlag( Persistent );

    Bitmap* Buffer;
    ...
};
```

`ReadFrom()` wird entsprechend generiert. Bei der Generierung von `Members()` wird nur der manuelle Anteil ersetzt. Eine von vornherein auf MopGen aufbauende Implementierung würde die Hilfsmakros `T...()` ersetzen. Ein Defizit hat der mit MopGen generierte Code gegenüber dem halbmanuell erzeugten: Falls ein Pointer als Vektor interpretiert werden soll, ist es mit MopGen nicht möglich, die Größe des Vektors anzugeben. Stattdessen kann an diesen Stellen ein Behälter-Objekt verwendet werden.

```
void name::Members(AccessMembers *ac)
{
```

```
    _accessMembers= ac;
    ScanMembers(
    $ForEachField
    {
        $If ( $FieldIsObject )
        { T( $FieldName ) }
        $Else $If ( $FieldIsPointer )
        {
            $If ( $FieldFlagIsSet( Vector ) )
            { TV( $FieldName, -1 ) }
            $Else $If ( $FieldIsCharType )
            { TS( $FieldName ) }
            $Else
            { TP( $FieldName ) }
        }
        $Else $If ( $FieldIsArray )
        {
            $If ( $FieldIsCharType )
            { TS( $FieldName ) }
            $Else
            { TP( $FieldName ) }
        }
        $Else $If ( $FieldIsComposite )
        {
            ... // T?P()-Fälle behandeln
        }
        $Else
        {
            $Warning( Typ von $Fieldname nicht für Klassenvariablen erlaubt)
        }
    }
}
```

13.4 Object-I/O á la Grossman

Die hier vorgestellte Implementation mit MopSpec hat gegenüber dem Original zwei Einschränkungen:

- Im Original wird das Header-File geändert; da MopGen nur neuen Code erzeugen kann, müssen die Deklarationen für die Ein-/Ausgabe-Methoden manuell in das Header-File eingefügt werden, z.B. per Makro.
- Der Original-Generator wird durch #pragma-Anweisungen gesteuert; MopSpec bietet stattdessen die Steuerung über das Ein- und Ausschalten von Flags (SET_..._FLAG()/RESET_..._FLAG()).

Der Code für die Ein- und Ausgabe der Exemplarvariablen ist nicht vollständig ausgeführt. Eine vollständige Spezifikation dafür findet sich in Abschnitt 12.7.

```
void $ClassName::flattenEntity( MG_baseFormatOut& strm )
{
    strm << classId << " $ClassName";          // Id und Klassennamen ausgeben
    if ( SaveEntityMap.Registered( this ) )    // wenn Objekt schon ausgegeben
    {                                          // wurde, ...
        strm << SaveEntityMap.GetId( this );   // nur Object-Id ausgeben
    }
    else
    {
        Id& objId = SaveEntityMap.Register( this ); // sonst neue Id vergeben
        strm << objId;
        $ClassName dummy = $ClassName( strm, objId ); // und Objekt mit Output-
                                                      // Konstruktor ausgeben
    }
}

// output Constructor
$ClassName::$ClassName( MG_baseFormatOut& strm, const Id& theObject )
```

```
    :
$ForEachBase
{
    $BaseName( strm, theObject )    // der Stream und das auszugebende Objekt
    $If ( $Not $LastBase )          // müssen explizit an die Basisklassen-
    { , }                           // Output-Konstruktoren übergeben werden
}
{
    $ClassName* OutObj = theObject.GetThisPtr(); // OutObj ist das auszugebende
    $ForEachField                                // Objekt
    {
        $If ( $FieldIsPointer )
        {                           // referenzierte Objekte ausgeben
            if ( $FieldName )
                OutObj->$FieldName->flattenEntity( strm );
            else
                strm.OutputNullPtr();
        }
        $Else $If( ... )
        {
            // Der Code für die anderen Fälle ist analog zum
            // Object-I/O des Beispiel-MOP (12.7)
        }
        $Else
        {   // der einfache Fall
            strm << p->$FieldName;
        }
    }
}

$ClassName* $ClassName::unflattenEntity( MG_baseFormatIn& strm )
{
    MG_ClassId ClassId;
    Id ObjectId;

    strm >> ClassId;
    if ( ClassId = NullId )
        return NULL;
    strm >> ObjectId;
    if ( retp = ReadEntityMap.GetAdr( ObjectId ) )
        return ($ClassName*)retp;
    CreateFct FctPtr = MakeFctMap[ClassId];     // Zeiger auf Create-Funktion
                                                // für "virtuellen Konstruktor"
    return ($ClassName*) ((*FktPtr)( ObjectId, strm ));
}

            // die Adressen der makeNew-Methoden sind im Array
            // MakeFctMap abgelegt
$ClassName* $ClassName::makeNew( Id objId, MG_baseFormatIn& strm )
{
    return new (objId ) $ClassName( strm );
}

            // new wird überladen, damit jedes gelesene Objekt VOR
            // dem Einlesen, also vor dem Konstruktoraufruf
            // registriert werden kann. Andernfalls droht die
            // Endlosschleife, wenn ein Objekt einen Zeiger auf
            // sich selbst enthält
void* $ClassName::operator new ( size_t size, Id objId )
{
    void *p = malloc( size );
    ReadEntityMap.Register( p, id );
    return p;
```

```
}

// Einlese-Konstruktor analog zum Ausgabe-Konstruktor
$ClassName::$ClassName( MG_baseFormatIn& strm )
    :
$ForEachBase
{
    $BaseName( strm )
    $If ( $Not $LastBase )
    { , }
}
{
    $ForEachField
    {
        $If ( $FieldIsPointer )
        {
            $FieldName= $FieldBaseType::unflattenEntity();
        }
        $Else $If( ... )
        {
            // handle other cases:
            // arrays, strings, etc
        }
        $Else
        {   // default case
            strm >> $FieldName;
        }
    }
}
```

13.5 Das Metadata System

Die folgende Spezifikation unterscheidet sich in dreierlei Hinsicht vom Original:

- Es wird keine Aufzählungstyp-Meta-Information erzeugt. Hier müßte MopSpec noch entsprechend erweitert werden.
- Es werden für zusammengesetzte Typen keine eigenen Typ-Metaobjekte erzeugt. Dies ist auch nicht mehr erforderlich, da die Typdeklarationen von MopGen vor der Generierung in Normalform gebracht werden.
- Die Basisklasseninformation ist eine Liste (für Mehrfachvererbung).

Die Meta-Information für Built-In-Typen ist in der Metadata System-Bibliothek enthalten und muß daher nicht generiert werden.

```
static $ClassName::GetVMeta()
{
    if ( !cvPVMeta->GetIsInitialized() )
    {
        int i=0;
        $ForEachField
        {
            cvPVMeta->InitializeVarOffset( i++,
                (int)((long)&$FieldName - (long)this ) );
        }
    }
    return cvPVMeta;
}

static InitVar InitVar$ClassName[] =
{
  $ForEachField
  {
    {
        "$FieldName",                          // variable name
```

```
        $If( $FieldIsObject )                   // pointer level
        { 0, }
        $Else $If( $FieldIsPointer )
        { 1, }
        $Else $If( $FieldIsDoublePointer )
        { 2, }
        $Else
        { -1, $Warning( Unknown pointer level )}
        $If( $FieldIsArray )                    // array size
        { $FieldArraySize, }
        $Else
        { -1 }
        $If( $FieldIsPublic )                   // protection mode
        { Public, }
        $If( $FieldIsProtectec )
        { Protected, }
        $If( $FieldIsPrivate )
        { Private, }
        VMETA_$FieldBaseType,                   // ref. to type metaobject
        -1                                      // offset (is initialized
                                                // at runtime)
    }
  }
}

static InitBases InitBases$ClassName[] =
{
    $ForEachBase
    {
        VMETA_$BaseName,
    }
    VMETA_NULL
}

static VMetaInfo VMetaInfo$ClassName{
    sizeof( $ClassName ),                           // object size
    FMT_OBJECT,                                     // format as object
    InitBases$ClassName,                            // list of bases
    sizeof( InitVar$ClassName / sizeof( InitVar ) ) // no. of data members
    InitVar$ClassName
}

VMeta VMeta$ClassName {
    "$ClassName",           // name
    0                       // pointer level is always 0
    &VMetaInfo$ClassName
};

VMeta* $ClassName::cvPVMeta = &VMeta$ClassName;
```

13.6 Das Open C++

Zur Beschreibung der Implementierung s. auch 6.8.

```
$If( ClassFlagSet( reflective ) )
{
class refl_$ClassName : public $ClassName,      // neue Klasse deklarieren
    $If( ClassFlagSet( VerboseMetaObj ) )
    { public VerboseMetaObj }                   // sie erbt von der ausgewählten
    $Else                                       // Meta-Klasse
    { public MetaObj }
{
    static MethodId idarray[];

    $ForEachMethod
    {
        $MethodRet $MethodName$MethodSignature;
        virtual void $MethodName_call( ArgPac& args, RetPac& reply );
    }
};

static MethodId refl_X::idarray[] =     // im idarray werden die Aufruf-Stubs
{                                       // gespeichert
  $ForEachMethod
  {
    { "$MethodName", $MethodNo, &$MethodName_call },
  }
  { "", -1, NULL }
}


$ForEachMethod
{
    // die überladenen Methoden packen die Argumente ein
    // und rufen Meta_MethodCall auf
    int refl_X::$MethodName$MethodSignature
    {
        ArgPac args;
        ForEachParam
        {
            args.Add$ParamTypeString( &$ParamName );
        }
        RetPac reply;
        Meta_MethodCall( idarray[1], StdMethod, args, reply );
        return reply.GetAs$RetTypeString();
    }

    // die Methoden-Stubs packen die Argumente wieder aus
    // und rufen die Methode der Originalklasse auf
    void refl_X::$MethodName_call( ArgPac& args, RetPac& reply )
    {
        reply.SetAs$RetTypeString(
            $BaseName( 1 )::$MethodName(
                $ForEachParam
                {
                    args[$ParamNo].GetAs$ParamTypeString()
                    $If( $Not $ParamIsLast )
                    { , }
                }
            )
        );
    }
}
```

14 Anhang C: Literaturverzeichnis

[BKS92] Frank Buschmann, Konrad Kiefer und Michael Stal.
A Runtime Type Information System for C++.
In TOOLS 7, 1992, S. 265-274

[Cardelli89] Luca Cardelli.
Typeful Programming.
Technischer Report Nr. 45, Systems Research Center of the Digital Equipment Corporation, Palo Alto, 1989

[CarWeg85] Luca Cardelli und Peter Wegner.
On Understanding Types, Data Abstraction, and Polymorphism.
Computing Surveys, Vol. 17, No. 4, Dec. 1985, S. 471-522

[ChiMas93] Shigeru Chiba und Takashi Masuda.
Designing an Extensible Distributed Language with a Meta-Level Architecture.
In ECOOP 1993 Conf. Proc, S. 482-501

[Cointe87] Pierre Cointe.
Metaclasses are First Class: the ObjVlisp Model.
In OOPSLA'87 Proceedings, S.156-167

[Gamma92] Erich Gamma.
Objektorientierte Softwareentwicklung am Beispiel von ET++.
Springer Verlag, 1992

[GolRob89] Adele Goldberg und David Robson.
Smalltalk-80: The Language.
Addison Wesley Publishing Company, 1989

[Großman93] Mark Großman.
Object I/O and runtime type information via automatic code generation in C++.
In Journal of Object-Oriented Programming, July-August 1993, S. 34-42

[HWPh80] Karlfried Gründer und Joachim Ritter(†) (Hrsg.).
Historisches Wörterbuch der Philosophie, Bd. 5.
Wissenschaftliche Buchgesellschaft Darmstadt, 1980

[JohPal93] Richard Johnson und Murugappan Palaniappan.
MetaFlex: A Flexible Metaclass Generator.
In ECOOP 1993 , S. 502-527

[KBSW94] Thomas Kofler, Walter Bischofberger, Bruno Schäffer, André Weinand.
A Poor Man's Approach to Dynamic Invocation of C++ Member Functions.
Technical Report of the UBILAB (Information Technology Laboratory of the Union Bank of Switzerland), 1994

[KRB91] Gregor Kiczales, Jim de Rivières und Daniel G. Bobrow.
The Art of the Metaobject Protocol.
The MIT Press, Cambridge, Massachusetts, 1991

[Maes87] Pattie Maes.
Concepts and Experiments in Computational Reflection.
In OOPSLA’87 Proceedings, S.147-155

[Meyer92] Bertrand Meyer.
Eiffel: The Language.
Prentice Hall International (UK) Ltd., 1992

[MeyNer90] Bertrand Meyer und Jean-Marc Nerson.
Eiffel: the Libraries.
Interactive Software Engineering Inc., 1990

[Paepcke93] Andreas Paepcke.
Object-Oriented Programming: The CLOS Perspective.
The MIT Press, Cambridge, Massachusetts, 1993

[PalSch91] Jens Palsberg und Michael I. Schwartzbach.
Three Discussions on Object-Oriented Typing.
In ECOOP'91 Proceedings, S. 31-38

[PWJ92] Peter-Alexander Pauw, Ronald Werring und Angeliqe Jansen.
An operational metadata system for C++.
In TOOLS (Technology of Object-Oriented Languages and Systems)
USA 1992 Conf. Proc., S. 215-223

[Riehle95] Dirk Riehle.
How and Why to Encapsulate Class Trees.
Submitted to OOPSLA’95.

[Siberski94] Wolf Siberski.
Garbage Collection in C++.
Technischer Report der Universität Hamburg, Fachbereich Informatik.

[Stroustrup94] Bjarne Stroustrup.
The Design and Evolution of C++.
Addison Wesley Publishing Company, 1994

[Wegner87] Peter Wegner.
Dimensions of Object-Based Language Design.
In OOPSLA’87 Proceedings, S.168-182

[WP95] ANSI/ISO Standards Committee X3J16, WG21.
Working Paper for Draft Proposed International Standard for
Information Systems— Programming Language C+ +.
electronically published, 1995