Entwicklungsunterstützung für persistente strukturierte Systeme

Diplomarbeit

eingereicht bei

Prof. Dr. Joachim W. Schmidt
Arbeitsbereich Softwaresysteme
Technische Universität Harburg

und

Dr. Martin Lehmann
Universität Hamburg

von
Christian Koch
Moorkamp 6c
22844 Norderstedt

24. Juni 1997

Danksagung

An dieser Stelle möchte ich allen Menschen danken, die zum Gelingen dieser Arbeit beigetragen haben. Besonderer Dank gilt meinen Eltern, die mir das Studium der Informatik ermöglicht und mich zu jeder Zeit voll unterstützt haben.

Für seine unzähligen Ideen, Vorschläge und Kommentare und für die insgesamt sehr gute Betreuung danke ich Gerald Schröder. Weiterhin danke ich Herrn Prof. Dr. Joachim W. Schmidt und Herrn Dr. Martin Lehmann dafür, daß sie mir die Möglichkeit gegeben haben, diese Arbeit zu schreiben. Darüber hinaus gebührt ihnen Dank für wertvolle Ratschläge und Anmerkungen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit Unterstützungssystemen für die Entwicklung großer Softwaresysteme. Es wird besonders die Klasse der persistenten Softwaresysteme betrachtet, in deren Zentrum traditionell häufig Datenbanken und Anwendungstransaktionen stehen und für die Datenbankprogrammiersprachen (*persistent programming languages*) zunehmend an Bedeutung gewinnen (Pascal/R, DBPL, Tycoon, Napier88, P-Java). Diese großen Systeme zeichnen sich besonders dadurch aus, daß sie eine sehr große Anzahl von Codezeilen umfassen, die von vielen Entwicklern geschrieben und gewartet werden. Ein weiteres Charakteristikum ist die Lebensdauer, die in einem Zeitraum von mehreren Jahrzehnten angesiedelt werden kann.

Der Prozeß der Softwareentwicklung, auch als Lebenszyklus bezeichnet, reicht von Problemanalyse über Entwurf und Implementierung bis hin zu Installation und Wartung und umfaßt eine Menge an Methoden, Aktivitäten und (Zwischen-)Ergebnissen. Stellvertretend für die Vielzahl an Modellen wird hier das weitverbreitete Wasserfallmodell aus [Royce 1970] herangezogen (s. Abbildung 1.1), aber auch andere Modelle wie das Spiralmodell [Boehm 1986] sowie zahlreiche Varianten [Comer 1997] weisen die hier zu betrachtende Charakteristik auf. Aus der Anordnung der Phasen im Wasserfallmodell ist zu erkennen, daß die Implementierung wie andere Phasen des Modells auch direkte Auswirkungen auf die nachfolgenden Phasen hat. Dies ist an dem in Abbildung 1.1 dargestellten Modell anhand der Abfolge der Phasen, der Verzahnung und des wiederholten Eintritts in die Phasen zu sehen und findet sich in ähnlicher Form in anderen Modellen wieder. Beim Übergang von einer Phase zur nächsten sollten die dabei entstehenden Reibungsverluste bei der Transformation des Ergebnisses einer Phase in die darauf folgende – z.B. durch Umsetzen der Entwurfseinheiten in konkrete Konstrukte der Programmiersprache – möglichst gering gehalten werden, um die Komplexität des gesamten Entwicklungsprozesses nicht weiter zu erhöhen; das gilt umso mehr, wenn die Phasen mehrfach in wiederholten Iterationsschritten durchlaufen werden. Vielmehr können die Kosten niedrig gehalten werden, die in den nachfolgenden Phasen, insbesondere der Wartung anfallen, wenn die Entwurfseinheiten eine adäquate Entsprechung in der Implementierung haben. Auch die durch Änderungen entstehenden Kosten, die durchschnittlich 60% der Gesamtkosten eines Softwaresystems ausmachen [Sommerville 1995], können auf diese Weise gesenkt werden.

Softwaresystemen, insbesondere großen, ist eine durch viele Faktoren beeinflußte Komplexität inhärent [Booch 1994; Brooks 1995; Brown et al. 1992]. Zu diesen Faktoren zählen z.B.

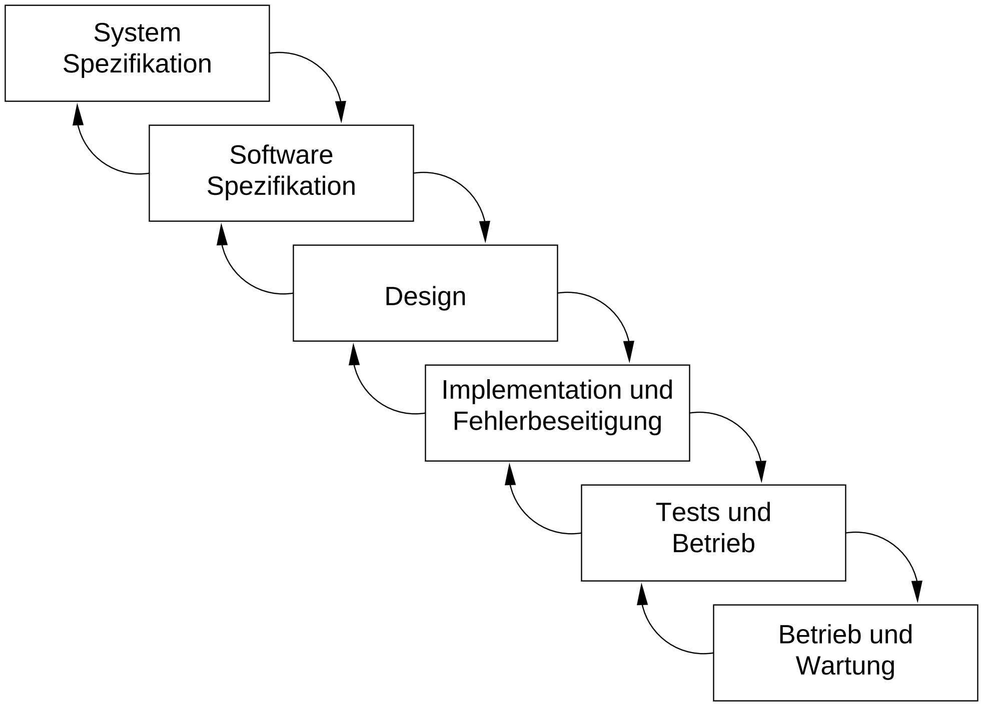


Abbildung 1.1: Wasserfallmodell (angelehnt an [Royce 1970]).

die große Menge geschriebenen und zu schreibenden Codes, dessen interne Abhängigkeiten und dessen Verständlichkeit, die durch fehlende, unzureichende oder nicht auf dem neuesten Stand befindliche Dokumentation weiter erschwert wird. Die folgenden Punkte verschärfen das Problem der Softwareentwicklung weiter [Brown et al. 1992]:

- Es besteht die Schwierigkeit, Anforderungen zu benennen und stabil zu halten.
- Die (oft negativen) Auswirkungen von Änderungen auf das Gesamtsystem können schwer abgeschätzt werden.
- Die komplexen und mehrdimensionalen Strukturen von Programmen lassen sich schwer visualisieren.
- Es existiert keine Domäne von Theorien, aus denen der Programmierer auswählen könnte, vielmehr wird der Softwareprozeß von der ständigen Entwicklung neuer Theorien begleitet.

Diese schon in der Entwurfsphase auftretende Problematik der Komplexität setzt sich in der Implementierung und in den weiteren Phasen des Entwicklungsprozesses fort. Entwickler müssen daher entsprechende Unterstützung auch in der Implementationsphase erfahren, um dieser Komplexität zu begegnen. Allgemeine Prinzipien dafür sind [Balzert 1982]:

Abstraktion: Hierunter versteht man das Absehen vom Konkreten. Sie ermöglicht das Erkennen allgemeiner Charakteristika. Mit Abstraktion wird das Wesentliche vom Unwesentlichen getrennt, und es lassen sich Objekte ordnen, klassifizieren und gewichten.

Strukturierung: Die Struktur eines Systems gibt dessen Charakter wieder. Sie vernachlässigt Details und enthält nur wesentliche Merkmale. Die Struktur eines Entwurfs oder Programms trägt zur Verständlichkeit bei und erleichtert es, sowohl den Entwicklern selbst als auch anderen, sich einzuarbeiten. Darüber hinaus erhöht eine gute Struktur die Änderungsfreundlichkeit und die Wartbarkeit, hat also direkten Einfluß auf Phasen, die sich an die Implementierung anschließen.

Hierarchisierung: Eine Hierarchie ordnet Elemente eines Systems bestimmten Stufen oder Rängen zu. Das Prinzip der Hierarchisierung hängt stark mit dem Prinzip der Strukturierung zusammen, da Hierarchien auch zur Strukturbildung beitragen. Daher erhöht es ebenfalls die Verständlichkeit und Wartbarkeit, wobei Hierarchien in unterschiedlichen Formen, etwa netz-, baum- oder schichtenorientiert, auftreten können.

Modularisierung: Die Zerlegung eines Systems in kleine, überschaubare, voneinander weitgehend unabhängige Teile nennt man Modularisierung. Auch sie steht in engem Zusammenhang mit der Strukturierung, da diese Zerlegung in Einheiten schon zu einer Struktur führt. Neben den dort genannten Gesichtspunkten hat Modularisierung insbesondere den Vorteil der Änderungsfreundlichkeit und durch die Lokalisierung von Daten und Algorithmen auch den Vorteil der besseren Überprüfbarkeit.

Lokalität: Hierunter versteht man das Vorhandensein relevanter Informationen an einem Ort. Das Prinzip der Modularisierung führt bereits zu Lokalität. Das Vorhandensein relevanter Information an einem Platz kann jedoch auf vielen weiteren Ebenen des Entwurfs und der Implementierung erreicht werden und führt dann zu Verständlichkeit und Lesbarkeit. Weiterhin erleichtert es die Wartung.

Wiederverwendbarkeit: Die Verwendung bereits vorhandener Software-(Teil-)Produkte verspricht je nach Grad eine Verkürzung der Entwicklungsdauer und damit eine Reduzierung der Kosten. Die für aufwendige Neuentwicklungen benötigt Zeit kann gespart oder für die Verbesserung bestehender Software eingesetzt werden.

Aufgabe einer Entwicklungsumgebung und einer Programmiersprache, aber auch einer Entwurfsmethodik muß es sein, diese Prinzipien zu unterstützen und die Erfüllung in höchstem Maße zu gewährleisten.

Der Schwerpunkt dieser Arbeit liegt auf der Unterstützung der Implementierung und des Entwurfs. Die Aufgabe der Implementierung ist es, ein ablauffähiges Programm zu erstellen, das der in den vorangegangenen Phasen erstellten Spezifikation entspricht. Besonderes Augenmerk sollte bei der Implementierung darauf gelegt werden, die in der Entwurfsphase herausgebildeten Einheiten, meist Module oder Klassen, möglichst direkt, nachvollziehbar und effizient umzusetzen. Daneben spielen bei der Implementierung weitere Aspekte eine Rolle, wie z.B. Anfertigung adäquater Dokumentation, Verwendung bestimmter Algorithmen usw., die in dieser Arbeit jedoch nicht betrachtet werden. Der in dieser Arbeit gewählte Ansatz hat zum Ziel, unabhängig von der gewählten Entwurfsmethodik, Unterstützung für deren programmiersprachliche Umsetzung zu bieten.

Hier wird daher der Ansatz verfolgt, eine Systemarchitektur bereitzustellen, die die flexible Anbindung von Werkzeugen ermöglicht. Das wird beispielhaft anhand der Realisierung eines Werkzeugs zur Verwaltung von Softwareeinheiten vorgeführt. Weiterhin wird mit TLMIN (*Tycoon Language minimal*) eine Sprache vorgestellt, die durch ihre leistungsfähigen Mechanismen zur Systemstrukturierung die Verwendung obiger Prinzipien fördert.

Wie Brooks in [Brooks 1995] ausführt, existiert keine Methodik zur Softwareentwicklung, die alle Probleme der Komplexität, Beherrschbarkeit usw. löst. Auch von den in dieser Arbeit vorgestellten Konzepten wird das nicht behauptet. Es besteht jedoch die Hoffnung, zumindest zu einer Verbesserung der existierenden (noch unzureichenden) Methoden beitragen zu können.

Neben den hier betrachteten Aspekten der Entwicklungsunterstützung lassen sich weitere Faktoren nennen, wie z.B. die Motivation und Güte der Ausbildung der Entwickler oder die zur Verfügung stehende technische Ausrüstung, die die Softwareentwicklung unterstützen. Sie werden an anderer Stelle entsprechend gewürdigt [Reifer 1993; van der Veer et al. 1988].

1.1 Aufgabenstellung

Die vorliegende Arbeit gliedert sich in zwei Bereiche, die den zwei betrachteten Aspekten (Entwicklungsumgebung, Sprachmechanismen) der Entwicklungsunterstützung entsprechen. Der erste Teil behandelt das Thema Entwicklungsumgebungen. Es handelt sich hierbei um Werkzeuge, die nicht Teil der Programmiersprache sind, sondern durch externe Programme, die meist mit weiteren Programmen zu Entwicklungsumgebungen zusammengefaßt sind, realisiert werden. Es wird zunächst ein allgemein akzeptiertes Referenzmodell für Entwicklungsumgebungen diskutiert [Brown et al. 1992; NIST 1993], um anschließend an zwei konkreten Beispielen in Form von kommerziell verfügbaren Entwicklungsumgebungen Untersuchungen durchzuführen. Das Hauptaugenmerk liegt dabei auf der Anbindung der Werkzeuge und der Unterstützung der Systemstrukturierung.

Der zweite Bereich widmet sich den Strukturierungsmechanismen von Programmiersprachen. Nach der Betrachtung von praxisrelevanten Sprachen werden die im TLMIN System realisierten Strukturierungsmechanismen vorgestellt. Es handelt sich dabei um Strukturierungsprimitiven auf Sprach- und Werkzeugebene. Grundlage hierfür bildet das am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg entwickelte Tycoon System und hier speziell die Unterfamilie der TLMIN Systeme. Auf dieser Basis werden in dieser Arbeit ein wiederverwendbarer Modulmanager für Softwareeinheiten entwickelt und die im System notwendigen Voraussetzungen geschaffen. Hierbei handelt es sich in erster Linie um eine Compilerschnittstelle. Neu entwickelte Mechanismen der Systemstrukturierung, die in die Sprache TLMIN bzw. den Übersetzer eingebaut sind, unterstützen die Entwicklung auf Sprachebene. Sie erlauben die explizite Kontrolle der Sichtbarkeit von Bezeichnern und gestatten es, mit Umgebungen (*environments*) zu arbeiten. Umgebungen können zusammen mit der Compilerschnittstelle als Grundlage für eine interaktive Schnittstelle dienen, indem sie die übersetzten Bindungen aufnehmen. Darüber hinaus lassen sich mit Umgebungen netzwerkartige Strukturen aufbauen, die z.B. für die Organisation mehrerer Entwicklungsbereiche benutzt werden können.

Grundlage für den TLMIN bezogenen Teil bilden andere Arbeiten zu TLMIN und TL Systemen. In [Matthes und Schmidt 1992], [Matthes 1993] und [Matthes und Müßig 1993] werden TL (*Tycoon Language*) , der Vorläufer von TLMIN, und die grundlegende Systemarchitektur vorgestellt. [Mathiske et al. 1993] beschreibt verfügbare Bibliotheken, die auch als Grundlage für die im Rahmen dieser Arbeit angefertigte Implementierung dienen. Syntaxerweiterungen stellt [Schröder 1994] vor, während in [Geisler 1995] reflektive Mechanismen beispielhaft in die TL Systeme eingeführt werden. In [Bremer 1996] werden Ansätze und Lösungen für die Typüberprüfung gegeben. Besonders wird auf die eingehenden Erfahrungen bei der Untersuchung der Modulverwaltung der TL Systeme in [Koch 1996b] zurückgegriffen.

1.2 Gliederung

Die vorliegende Arbeit gliedert sich in die folgenden Kapitel:

- Im Anschluß an dieses Kapitel wird in Kapitel 2 die begriffliche Grundlage in bezug auf Entwicklungsaufgaben gelegt, indem ein Referenzmodell für Entwicklungsumgebungen in Teilen vorgestellt wird. Daran schließt sich die Untersuchung zweier in der Praxis eingesetzter Umgebungen an. Hier wird der Schwerpunkt auf die Integration der durch die Programmiersprache gegebenen Strukturierungskonzepte mit den durch die Entwicklungsumgebung bereitgestellten Werkzeugen gelegt.

- Den Ausgangspunkt für die Untersuchung von sprachlichen Mitteln zur Systemstrukturierung bildet die Vorstellung von Mechanismen, wie sie in gängigen Programmiersprachen gefunden werden. Zur Darstellung dieser Mechanismen dient eine Notation zur Beschreibung von Sichtbarkeitsbereichen, wie sie in Kapitel 3 eingeführt wird.

- Als Implementierungsplattform wird die TLMIN Architektur gewählt. Kapitel 4 führt zunächst grundlegende Begriffe ein, um dann den Übersetzer des TLMIN Systems in seinen Grundzügen zu beschreiben. An die Darstellung der reflektiven Compilerschnittstelle schließt sich die Diskussion des für Werkzeuge der Entwicklungsumgebung verwendeten Ansatzes zur Anbindung an das System an, der in der Realisierung der Modulverwaltung konkretisiert wird.

- Kapitel 5 legt die Strukturierungsmittel des TLMIN Systems dar. Großen Raum nimmt dabei die Darstellung der Basismechanismen ein. Aufbauend darauf verdeutlicht ein Modulkonzept die Verwendung dieser Basisprimitiven. Ein weiteres übergeordnetes Konzept wird mit den Umgebungen vorgestellt.

- Der Ausblick auf weiterführende Forschungsgebiete beschließt diese Arbeit. Mit Hinweisen auf die kommerzielle Relevanz der untersuchten Konzepte und mit einem Überblick über die begleitend durchgeführte Programmierarbeit endet Kapitel 6.

- Die Anhänge bilden für den interessierten Leser einen Ausgangspunkt für eine tiefergehende Studie insbesondere der Implementierungsarbeiten. Sie dienen außerdem der Vervollständigung einiger Ausführungen des Haupttextes.

Kapitel 2

Entwicklungsumgebungen

In diesem Kapitel wird die begriffliche Grundlage für die Erörterungen in den nachfolgenden Kapiteln gelegt. Ein Schwerpunkt der Betrachtungen ist der Aspekt der Systemstrukturierung und hier im besonderen das Zusammenwirken von Basisdiensten und externen Programmen bzw. die Unterstützung, die durch externe Programme geleistet werden kann. Externe Programme, im Zusammenhang mit Entwicklungsumgebungen auch Werkzeuge genannt, und die Dienste, die sie anbieten, werden hier im Unterschied zu Diensten gesehen, die direkt von einer Programmiersprache (z.B. Systemstrukturierung) bzw. einem Basissystem (z.B. Dateiverwaltung) geleistet werden. Sie benötigen zur Erfüllung ihrer Aufgabe eine Reihe von grundlegenden Diensten. Z.B. muß ein Werkzeug, welches die Übersetzung von Modulen steuert, Zugriff auf einen Dienst haben, mit dessen Hilfe es auf Dateien zugreifen oder das Datum der letzten Veränderung ermitteln kann. Dieser Dienst kann auf unterschiedliche Weise bereitgestellt werden. Eine Bibliothek kann die entsprechenden Dienste in Form von Funktionen anbieten, andere Dienste können durch Systemaufrufe bereitgestellt werden, wieder andere durch externe Programme einer niedrigeren Schicht, meist des Betriebssystems. In letzerem Fall sind die Werkzeuge und die zugrundeliegenden Dienste selbständig, also unabhängig von einer Programmiersprache zu sehen.

In dem vorliegenden Kapitel wird zunächst anhand einer Referenzarchitektur für Entwicklungsumgebungen aufgezeigt, welche Dienste Werkzeugen zur Verfügung stehen. Anhand zweier konkreter Beispiele wird beschrieben, ob und wie diese Dienste Werkzeugen bereitgestellt werden. Am Beispiel der ABAP/4 Entwicklungsumgebung wird ferner im Vorgriff auf die weiteren Kapitel eine enge Verzahnung von Programmiersprache und Entwicklungsumgebung gezeigt und in diesem Zusammenhang auch die Verteilung der Strukturierungsaufgaben zwischen diesen betrachtet.

Der Begriff Entwicklungsumgebung selbst ist in der Literatur nicht eindeutig definiert. Informell wird in [NIST 1993] eine Entwicklungsumgebung als ein System gesehen, daß automatisierte Unterstützung für die Softwareentwicklung bereitstellt und die Prozesse, die hierfür notwendig sind, verwaltet. Es gibt darüber hinaus zahlreiche weitere Begriffe, die die gleiche oder eine ähnliche Bedeutung haben. Eine Auswahl dieser Begriffe, die alle die gleiche Thematik betreffen, ist nachfolgend gegeben:

- Computerunterstützte Software Entwicklungsumgebung (*Computer Aided Software Engi-*

neering Environment (CASEE))

- Integrierte Projektunterstützungsumgebung (*Integrated Project Support Environment (IP-SE)*)
- Integrierte computerunterstützte Softwareentwicklung (*Integrated Computer Aided Software Engineering (I-CASE)*)
- Integrierte Softwarefabrik (*Integrated Software Factory (ISF)*)
- Integrierte Softwareentwicklungsumgebung (*Integrated Software Engineering Environment (ISEE)*)
- Softwareentwicklungsumgebung (*Software Development Environment (SDE)*)
- Softwareentwicklungsumgebung (*Software Engineering Environment*)
- Unterstützungsumgebung (*Support Environment*)

Die Aufgaben einer Entwicklungsumgebung lassen sich wie folgt zusammenfassen. Entwicklungsumgebungen verwalten Informationen über Software (Quellcode, Entwurfsdaten, Testdaten usw.), über Projektressourcen (z.B. Entwicklerkapazitäten oder Rechnerkapazitäten) und über Organisation, Standards und Richtlinien, die bei der Softwareentwicklung Anwendung finden.

Der Fokus dieser Arbeit liegt auf der Betrachtung des ersten Punktes, Informationen, die unmittelbar mit dem Entwurfsprozeß in Verbindung stehen.

2.1 Referenzarchitektur für Entwicklungsumgebungen

In [NIST 1993] wird ein Referenzmodell für die Architektur einer Entwicklungsumgebung vorgestellt. Dabei werden diejenigen Dienste spezifiziert, die eine Entwicklungsumgebung bereitstellen sollte. Diese Dienste sind in Gruppen zusammengefaßt. Eine Grundstruktur wird vorgegeben, welche einen Architekturvorschlag darstellt, der die Berührungspunkte der einzelnen Dienstgruppen aufzeigt. Die folgenden Dienstgruppen sind in [NIST 1993] definiert:

- Objektverwaltungsdienste (*object management services*)
- Prozeßverwaltungsdienste (*process management services*)
- Kommunikationsdienste (*communication services*)
- Betriebssystemdienste (*operating system services*)
- Benutzerschnittstellendienste (*user interface services*)
- Sicherheitsbezogene Dienste (*policy enforcement services*)
- Administrationsdienste (*framework administration services*)

Abbildung 2.1 zeigt den prinzipiellen Aufbau der Referenzarchitektur, wobei die sicherheitsbezogenen Dienste und die Administrationsdienste nicht in dieser Darstellung enthalten sind. Besonders hervorgehoben werden soll hier die Stellung, die die Werkzeuge einnehmen. Sie setzen auf die Funktionalität auf, die von den übrigen Diensten bereitgestellt wird. Eine zentrale Position nehmen dabei zum einen die Kommunikationsdienste ein, die als Bindeglied zwischen allen anderen Diensten fungieren. Zum anderen stellt die Gruppe der Objektverwaltungsdienste eine weitere wichtige Komponente dar. Hier werden Objekte wie der Quellcode oder übersetzte Einheiten verwaltet und in Beziehung zueinander gebracht.

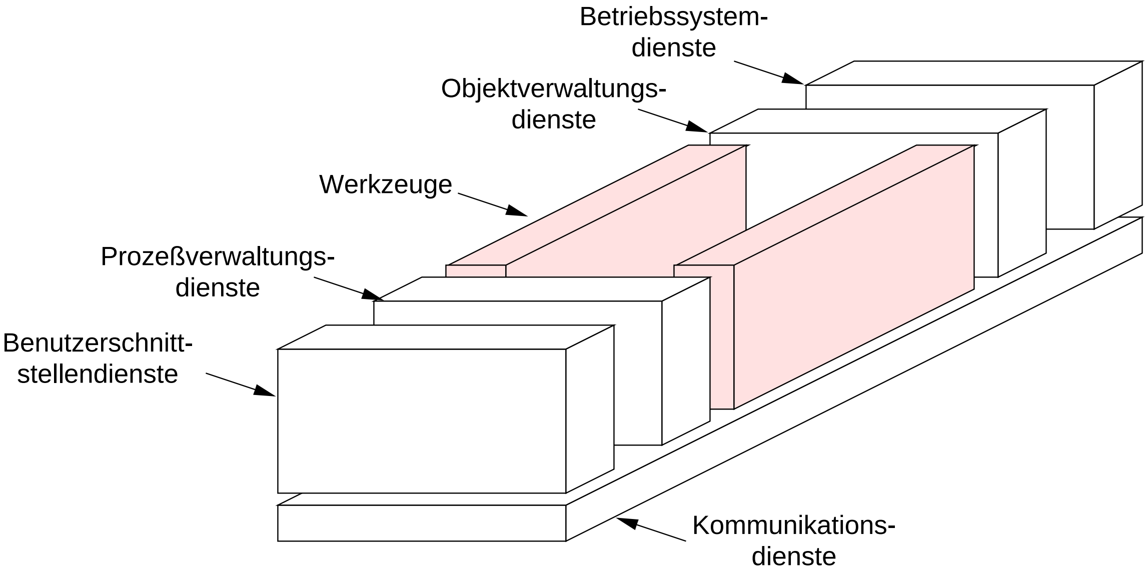


Abbildung 2.1: Überblick über eine Referenzarchitektur für Entwicklungsumgebungen.

Allgemein verwalten Entwicklungsumgebungen Informationen aus den folgenden drei Kategorien:

a) Information über die in der Entwicklung befindliche Software, wie z.B. Spezifikations-, Entwurfs-, Quellcode- und Testdaten.

b) Informationen über Ressourcen, wie z.B. die verfügbare Hardware und deren Leistungsfähigkeit sowie die zur Verfügung stehenden Entwickler, und die Zuordnung dieser Ressourcen zu bestimmten Aufgaben.

c) Informationen über die zugrundeliegende Unternehmensstruktur, (Unternehmens-) Standards und andere Richtlinien, die die Entwicklung von Software betreffen.

Wie in Abbildung 2.1 zu sehen, werden die Aufgaben einer Entwicklungsumgebung zum einen durch die bereits erwähnten Dienste erfüllt. Diese stellen aber zum anderen auch den Werkzeugen geeignete Mittel zur Verfügung, um diese Aufgaben zu erledigen, bzw. nutzen die Werkzeuge ihrerseits, um Dienste zu erbringen (Prozeßverwaltungsdienste und Benutzerschnittstellendienste).

In den nächsten Abschnitten werden von den vorstehend aufgeführten Dienstgruppen einige exemplarisch dargestellt. Die hierbei betrachteten Aspekte sind die Systemstrukturierung und die Anbindung von Werkzeugen. Für den ersten Aspekt ist nur die Dienstgruppe der Objektverwaltungsdienste relevant, da sie direkt mit den Softwareeinheiten wie z.B. Modulen und Klassen operiert.

Das vorgestellte Referenzmodell stellt lediglich Basisdienste z.B. für Werkzeuge bereit. Es werden deshalb die Möglichkeiten zur Integration von Werkzeugen betrachtet. In [NIST 1993] werden folgende Dimensionen beschrieben, auf denen Integration erreicht werden kann. Sie dienen als Anhaltspunkt für die Erörterung der weiteren Dienstgruppen.

Datenintegration (*data integration*): Hierunter versteht man die Möglichkeit, daß Werkzeuge Daten untereinander austauschen können. Dies kann z.B. erreicht werden, indem verschiedene Werkzeuge die gleiche Datenbank benutzen oder indem ein einheitliches Datenformat verwendet wird.

Kontrollflußintegration (*control integration*): Das bedeutet, daß die Funktionalität von Werkzeugen auf unterschiedliche Weise und in unterschiedlicher Reihenfolge kombiniert werden kann.

Präsentationsintegration (*presentation integration*): Hiermit ist die einheitliche Darstellung der Werkzeuge auf dem Bildschirm gemeint. Dies kann z.B. durch die Verwendung einer einheitlichen Bedienoberfläche erreicht werden.

Prozeßintegration (*process integration*): Die Funktionalität der Umgebung kann in vordefinierten Prozessen genutzt werden.

Grundstrukturintegration (*framework integration*): Diese Integrationsdimension bezieht sich darauf, inwieweit Werkzeuge sich in eine gegebene Grundstruktur einpassen. Sie können z.B. ähnliche Installationsmechanismen oder ähnliche Mechanismen zur Benutzerverwaltung verwenden.

Insbesondere für die in Kapitel 4 behandelte Entwicklungsumgebung des TLMIN Systems werden hier diejenigen Dienste gezeigt, die die Basisdienste für Werkzeuge bereitstellen. Das sind Betriebssystemdienste, Objektverwaltungsdienste und Kommunikationsdienste. Prozeßverwaltungsdienste und Präsentationsdienste nutzen wiederum die Funktionalität der Werkzeuge, um ihre Dienste anzubieten (s.a. Abbildung 2.1).

Bei der folgenden Schilderung ist zu beachten, daß die dargestellten Dienste nicht in ihrer Gesamtheit existieren müssen, um aufbauend darauf Werkzeuge einer Entwicklungsumgebung realisieren zu können. Vielmehr stellen sie eine Obermenge der benötigten Dienste dar. Vielfach wird ihre Funktionalität erst durch Werkzeuge implementiert. Dies ist u.a. darin begründet, daß die Dienste sehr nah an den vom Benutzer der Entwicklungsumgebung tatsächlich verwendeten Diensten sind.

2.1.1 Betriebssystemdienste

Die zu dieser Gruppe gehörenden Dienste stellen die von einem Betriebssystem üblicherweise verfügbaren Funktionen in einer konsistenten Weise bereit. Sie abstrahieren damit von den Details des konkreten Systems und fördern die Portabilität. Für die Anbindung bzw. Kommunikation von Werkzeugen oder für die Systemstrukturierung haben sie eine Bedeutung, wenn die Objektverwaltung auf Basis von Dateien vorgenommen wird. In diesem Fall korrespondiert die Einheit Datei direkt mit einer Softwareeinheit (Modul, Klasse, Schnittstelle o.ä.). Folgende Dienste sind in dieser Dienstgruppe zusammengefaßt:

Prozeßverwaltung (*operating system process management*): Verwaltung, also Erzeugen, Duplizieren oder Beenden von Betriebssystemprozessen.

Umgebungsdienste (*operating system environment service*): Prozesse laufen meist in einer Umgebung ab, die z.B. aus einer Menge von Variablen mit entsprechenden Werten bestehen kann. Zur Umgebung zählen aber auch Zeit- und Datumsinformation oder Obergrenzen für verfügbare Ressourcen wie z.B. Speicher oder Anzahl der offenen Dateien.

Synchronisationsdienste (*operating system synchronisation service*): Hierunter fällt die Funktionalität des Betriebssystems, Prozessen nach einer bestimmten Strategie Zeit zur Ausführung zuzuteilen.

Ein- und Ausgabedienste (*generalized input and output*): Darunter sind generelle Mechanismen zur Ein- bzw. Ausgabe von Daten auf physikalischen Geräten zusammengefaßt.

Dateiein- und Dateiausgabe (*file storage service*): Der Satz von Operationen ähnelt im wesentlichen denen der zuvor beschriebenen generellen Ein- und Ausgabedienste. Als Besonderheiten kommen hier Operationen zum Zugriff auf einen Verzeichnisdienst hinzu.

Signaldienst (*asynchronous event service*): Prozesse signalisieren unabhängig von sonstigen Kommunikationsdiensten bestimmte Ereignisse mit Hilfe von Signalen. Diese werden meist in Ausnahmesituationen, z.B. bei schweren Fehlern, die die weitere Ausführung unmöglich machen, eingesetzt.

Zeitdienste (*interval timing service*): Hierunter fallen Dienste zur Manipulation bestimmter Zeitmeßmechanismen des Betriebssystems.

Speicherdienste (*memory management service*): Dieser Dienst wird vom Betriebssystem meist nur bei der Anforderung von großen dynamischen Speicherblöcken direkt abgefragt.

Physikalischer Gerätedienst (*physical device service*): Geräte, die mit den bereits beschriebenen Diensten nicht angesprochen werden können oder eine Sonderbehandlung erfahren müssen, können über diesen Dienst verwaltet werden.

Ressourcenverwaltungsdienst (*operating system resource management service*): Unter diesem Begriff sind Dienste zusammengefaßt, die es ermöglichen, Ressourcen wie z.B. Sekundärspeichergrenzen (*quotas*) zu ändern.

In bezug auf die Systemstrukturierung sind hier lediglich die Ein- Ausgabedienste relevant. Sie sorgen dafür, daß die verwalteten Einheiten persistent abgelegt werden.

2.1.2 Objektverwaltungsdienste

Bei den Diensten in dieser Gruppe geht es um die Definition, Ablage, Wartung, Verwaltung und den Zugriff von bzw. auf Daten. In diesem Fall sind mit Daten alle Objekte gemeint, die in dem Softwareentwicklungsprozeß eine Rolle spielen. Hierzu gehören z.B. Module und Klassen und deren Beziehungen untereinander.

Die folgenden Absätze schildern die Bedeutung dieser Dienste für die Systemstrukturierung. Der Aspekt der Werkzeugunterstützung ist hier insofern betroffen, als eine gemeinsame Datenbasis als Grundlage für den Austausch von Objekten dienen kann und diese Dienste Objekte assoziieren können.

Metadatendienst (*metadata service*): Metadaten legen die Struktur der verwendeten Daten und evtl. auf ihnen definierte Nebenbedingungen fest. Üblicherweise wird das verwendete Datenmodell einer Entwicklungsumgebung, die eng mit einer Programmiersprache verbunden ist, durch diese Sprache festgelegt. Somit sind die Mechanismen zur Definition entsprechender Strukturen bereits in der Sprache enthalten.

Datenspeicherungsdienst (*data storage and persistence service*): Die Daten der Entwicklungsumgebung müssen persistent gespeichert werden. Das kann erfolgen, indem dieser Dienst auf entsprechende Dateiverwaltungsdienste der Gruppe der Betriebssystemdienste aufsetzt. Eine andere in Kapitel 4 betrachtete Möglichkeit ist die Verwendung eines persistenten Objektsystems, welches Persistenz in Form von transitiver Erreichbarkeit bereitstellt.

Beziehungsdienst (*relationship service*): Zwischen den Objekten der Entwicklungsumgebung bestehen Beziehungen. Diese können in vielerlei Gestalt auftreten, z.B. als Beziehungen, die durch Importe zwischen Quelldateien bestehen oder als Beziehungen, die zwischen zusammenzubindenden übersetzten Einheiten bestehen.

Namensdienst (*name service*): Objekte müssen, damit sie für Werkzeuge erreichbar sind, benannt werden. Dieser Dienst stellt ein einheitliches Benennungsschema für Objekte bereit.

Verteilungs- und Lokalisierungsdienst (*distribution and location service*): Bei der vorgestellten Referenzarchitektur wird nicht von einer zentralen Datenbasis ausgegangen. Dieser Dienst stellt deshalb Mechanismen bereit, um die physikalische Verteilung der Daten in ein logisches Schema für den Entwickler umzusetzen. Dieses logische Schema kann wiederum von der Verteilung vollständig abstrahieren.

Transaktionsdienst (*data transaction service*): Dieser Dienst stellt ähnlich dem Transaktionsbegriff aus dem Bereich der Datenbanken eine Möglichkeit bereit, logisch zusammengehörende Aktivitäten zu einer Einheit zusammenzufassen, die atomar und isoliert ausgeführt wird und dabei den Datenbestand in einem konsistenten Zustand hinterläßt.

Nebenläufigkeitsdienst (*concurrency service*): Der Zugriff auf die Objekte kann von mehreren Prozessen, Werkzeugen bzw. anderen Diensten simultan erfolgen. Dieser Dienst synchronisiert diese nebenläufigen Zugriffe.

Betriebssystemprozeßdienst (*os process support service*): Durch diesen Dienst wird die Möglichkeit geschaffen, auf Betriebssystemprozesse in der gleichen Weise wie auf die Objekte der Entwicklungsumgebung zuzugreifen.

Archivierungsdienst (*archive service*): Dieser Dienst speichert die Objektdaten auf tertiären Medien.

Backupdienst (*backup service*): Dieser Dienst legt Strategien zur Sicherung der Daten fest und bietet Möglichkeiten zum Wiedereinspielen gesicherter Daten nach einem Ausfall.

Ableitungsdienst (*derivation service*): Objekte können durch Transformation aus anderen Objekten hervorgehen. Der Ableitungsdienst spezifiziert die Regeln, nach denen die Transformation eines Objektes ausgeführt wird und mit welchen Werkzeugen das geschieht. Dieser Dienst entspricht weitgehend dem, was eine Modulverwaltung leistet.

Replikations- und Synchronisationsdienst (*replication and synchronisation service*): Dieser Dienst hat ausschließlich in einer verteilten Umgebung eine Bedeutung. Dort regelt er die Verteilung und Replikation der Objekte und deren Konsistenz.

Zugriffskontroll- und Sicherheitsdienst (*access control and security service*): Unterschiedliche Benutzer der Entwicklungsumgebung haben unterschiedliche Bedürfnisse an den Umfang an Daten, die sie benutzen. Darüber hinaus ist es notwendig, den Zugriff auf die Daten explizit kontrollieren zu können.

Funktionsanhangsdienst (*function attachment service*): Dieser Dienst verfolgt einen objektorientierten Ansatz, der es erlaubt, an ein Objekt Funktionen anzuhängen, die auf dem Objekt arbeiten. Bei diesen Funktionen kann es sich z.B. um Methoden handeln, die eine bestimmte Transformation auf dem Objekt durchführen.

Schemadienst (*common schema service*): Besonders wichtig für die Integration von Werkzeugen ist der Schemadienst. Er stellt ein einheitliches Schema für die verwalteten Daten bereit und legt somit die Struktur der Daten, Integritätsbedingungen sowie deren Namen fest. Weiterhin können in einem Schema die auf einem Objekt möglichen Operationen spezifiziert werden.

Versionierungsdienst (*version service*): Das ist ein Dienst, der direkt an den Benutzer der Entwicklungsumgebung gereicht werden kann. Er verwaltet aktuelle und historische Zustände der Objekte. Es können von einem Objekt parallel mehrere Versionen bestehen, die in einer gemeinsamen Version zusammengeführt werden oder, falls es sich z.B. um Versionen für unterschiedliche Betriebssystemplattformen handelt, nebeneinander existieren.

Dienst für zusammengesetzte Objekte (*composite object service*): Objekte können zu größeren Objekten aggregiert werden. Die hierfür und für den Zugriff auf die Komponenten notwendigen Operationen stellt dieser Dienst bereit.

Anfragedienst (*query service*): Neben Diensten, die direkt auf ein Objekt zugreifen, bietet der Anfragedienst die Möglichkeit, zunächst unbekannte Objekte durch Formulierung einer entsprechenden Anfrage zu lokalisieren.

Überwachungs- und Ereignisdienst (*state monitoring and triggering service*): In Abhängigkeit von bestimmten Zuständen löst dieser Dienst Aktionen aus. Ein weiteres Merkmal besteht in der Überwachung der Zustände.

Klassifikationsdienst (*data subsetting service*): Dieser Dienst muß von dem Dienst für zusammengesetzte Objekte unterschieden werden. Oft ist es sinnvoll, nicht die gesamte Menge der Objekte der Entwicklungsumgebung zu betrachten, sondern eine in sich geschlossene Untermenge.

Datenaustauschdienst (*data interchange service*): Zum Austausch der Daten z.B. mit einer anderen Entwicklungsumgebung stellt dieser Dienst Funktionen bereit.

Durch die Objektverwaltungsdienste wird die Systemstrukturierung in direkter Weise unterstützt. Dies geschieht zum einen durch die Möglichkeit, Objekte zu benennen und zusammenzufassen und zum anderen durch die Möglichkeit, Abhängigkeiten und Umformungen auf diesen Objekten zu definieren.

2.1.3 Kommunikationsdienste

Dienste, die in diese Gruppe fallen, ermöglichen den Werkzeugen und den Diensten anderer Gruppen, miteinander zu kommunizieren. Besonders die Aufgaben, die Werkzeuge erfüllen, können nicht immer in Isolation gesehen werden. Werkzeuge müssen, z.B. um den Bearbeitungszustand eines Objektes abzufragen, miteinander kommunizieren können. Zwar bieten die Objektverwaltungsdienste auch ein gewisses Maß an Kommunikation, indem sie die Objekte allen Werkzeugen zur Verfügung stellen und diese Werkzeuge somit über Zugriff und Modifikation dieser Objekte Informationen austauschen können. Bei den Kommunikationsdiensten handelt es sich jedoch um eine andere Qualität der Verständigung, die z.B. darauf abstellt, in Ausführung befindliche Werkzeuge durch entsprechende Mechanismen zu synchronisieren oder Werkzeuge durch die Signalisierung anderer Werkzeuge zu benachrichtigen, eine gestoppte Ausführung fortzusetzen.

In dieser Arbeit liegt der Schwerpunkt auf der Betrachtung von Werkzeugen, somit werden die hier angebotenen Dienste besonders unter dem Gesichtspunkt der Kommunikation von Werkzeugen dargestellt.

Dienst für gemeinsamen Datenzugriff (*data sharing service*): Dieser Dienst faßt die Funktionalität verschiedener anderer Dienste aus der Gruppe der Objektverwaltungsdienste zusammen, um den gemeinsamen Zugriff auf Daten zur Verfügung zu stellen.

Prozeßkommunikationsdienst (*interprocess communication service*): Der auch vom Betriebssystem zur Verfügung stehende Dienst zur Interprozeßkommunikation wird hier noch einmal bereitgestellt.

Netzwerkdienst (*network service*): Dieser Dienst umfaßt Operationen zum Ausführen entfernter Prozeduraufrufe und zum Ermitteln und Benutzen entfernter Ressourcen.

Nachrichtenvermittlungsdienst (*message service*): Dieser Dienst ermöglicht Prozessen oder Werkzeugen, Daten über Nachrichten auszutauschen. Z.B. lassen sich alternativ zum Zugriff auf gemeinsame Daten diese Daten auch mittels Nachrichten übertragen.

Ereignisbenachrichtigungsdienst (*event notification service*): Im Gegensatz zum Nachrichtenvermittlungsdienst sind die Nachrichten, die durch den Ereignisbenachrichtigungsdienst verschickt werden, an das Auftreten eines Ereignisses gebunden.

Wie die Betriebssystemdienste stellen auch die Dienste in dieser Dienstgruppe grundlegende Mechanismen bereit und bieten keine unmittelbare Unterstützung für die Systemstrukturierung.

2.2 Teamware und SCCS

Bei den in diesem Abschnitt untersuchten Softwarekomponenten handelt es sich um eine von Sun Microsystems vertriebene Entwicklungsumgebung. Sie unterstützt die Entwicklung in einem Team und bietet darüber hinaus eine grafische Benutzerschnittstelle für die angebotenen Funktionen.

Diese Entwicklungsumgebung wird hier exemplarisch für die Klasse der Entwicklungsumgebungen untersucht, die die folgenden zwei Eigenschaften aufweisen:

- Die Verwaltung der Softwareeinheiten findet auf Basis von Dateien statt, d.h. eine Softwareeinheit ist in einer Datei enthalten. Das impliziert zum einen die hohe Erreichbarkeit der Einheiten durch (auch nicht zur Entwicklungsumgebung gehörende) Programme und damit eine potentiell hohe Flexibilität gegenüber Erweiterungen bzw. Austausch von Werkzeugen. Zum anderen kann bei der Verwaltung im Dateisystem angenommen werden, daß keine Abfragesprache (z.B. SQL oder OQL) existiert oder ein allgemein zugängliches Datenmodell auf den Softwareeinheiten definiert wäre. Dies folgt unmittelbar daraus, daß die Objektverwaltungsdienste von den Betriebssystemdiensten übernommen werden. Die Dateien bzw. Softwareeinheiten sind somit semantisch in keiner Weise verbunden (Import- und Exportabhängigkeit), und es können keine unterschiedlichen Typen identifiziert werden (Quelldatei, Objektdatei usw.).

- Die Entwicklungsunterstützung erfolgt sprachunabhängig. Zwar existieren auch in dieser Entwicklungsumgebung sprachabhängige Teile, sie sind jedoch nicht fester Bestandteil eines Werkzeuges. Vielmehr können die Programme durch geeignete Parametrisierungskonzepte (Konfigurationsdateien, Kommandozeilenargumente) an die jeweile Sprache angepaßt werden.

Die Softwareentwicklungsumgebung von Sun Microsystems umfaßt eine Reihe von Werkzeugen für jeweils bestimmte Aufgaben. Sie sind nachstehend beschrieben. Einen Überblick über die Komponenten und deren Zusammenwirken gibt Abbildung 2.2. Einige der Werkzeuge sind unter der Bezeichnung SPARCworks/ProWorks, andere unter SPARCWorks/TeamWare zusammengefaßt[1]. Im folgenden wird ein Überblick der SPARCworks/ProWorks Werkzeuge gegeben:

Source Code Browser: Dieses Werkzeug dient dazu, während der Programmentwicklung den Überblick über ein Softwareprojekt zu behalten oder zu erlangen. Es gestattet, z.B. alle Vorkommen einer Variablen, Funktion usw. anzuzeigen. Weiterhin kann in einer baumartigen Struktur der Aufrufgraf (*call graph*) eines Programms dargestellt werden.

MakeTool: Dies ist ein grafisches Frontend zu *make*, einem Programm, welches basierend auf spezifizierten Abhängigkeiten zwischen Dateien Aktionen auf diesen oder anderen Dateien ausführt [Feldmann 1979]. Bei diesen Aktionen kann es sich z.B. um die Übersetzung einer Quelldatei in eine Objektdatei handeln. Der Ansatz der dateibasierenden Entwicklungsumgebung gestattet es aber auch, beliebige andere Programme, die auf Dateien arbeiten, über das MakeTool aufzurufen und in den Entwicklungsprozeß einzubinden.

Debugger: Wie das MakeTool besteht auch der Debugger aus einem grafischen Frontend, welches die Steuerung eines eigenständig ablaufbaren Programms, dem eigentlichen Debugger, übernimmt. Der Debugger erlaubt das schrittweise Durchlaufen bzw. beliebiges Anhalten und erneutes Ablaufen eines Programms und die Untersuchung von Variablenwerten, Deklarationen, Funktionsaufrufen auf dem Stack etc. zur Laufzeit. Zu diesem Zweck liest der Debugger die vom Übersetzer generierten Informationen aus der Objektdatei aus.

Performance Tuning: Es handelt sich hierbei um ein Werkzeug, welches es dem Programmierer erlauben soll, hinsichtlich einer vom Programm benötigten Ressource, etwa Prozessorzeit oder Speicherplatz, Engpässe zu ermitteln, um diese anschließend zu beseitigen. Während der Fehlersuche mit dem Debugger kann er Informationen über das laufende Programm sammeln, z.B. wieviel Prozessorzeit für die Abarbeitung bestimmter Programmteile verbraucht oder in welchem Teil wieviel Speicher angefordert wurde. Aufgrund der Ergebnisse dieses Prozesses ist es dann möglich, Änderungen am Programm durchzuführen.

Die Komponenten der Teamware-Umgebung werden nachstehend erläutert. Die Konzepte, die dem VersionTool und dem Codemanager zugrunde liegen, werden in den nächsten Abschnitten weiter untersucht.

[1] Die zugrunde gelegte Dokumentation ist in diesem Punkt nicht eindeutig. Das FileMerge-Werkzeug wird sowohl der SPARCworks/ProWorks als auch der SPARCworks/TeamWare zugerechnet [Microsystems]. Da es von der Funktionalität her eng mit dem VersionTool der SPARCworks/TeamWare Umgebung zusammenarbeitet, wird es auch dort behandelt.

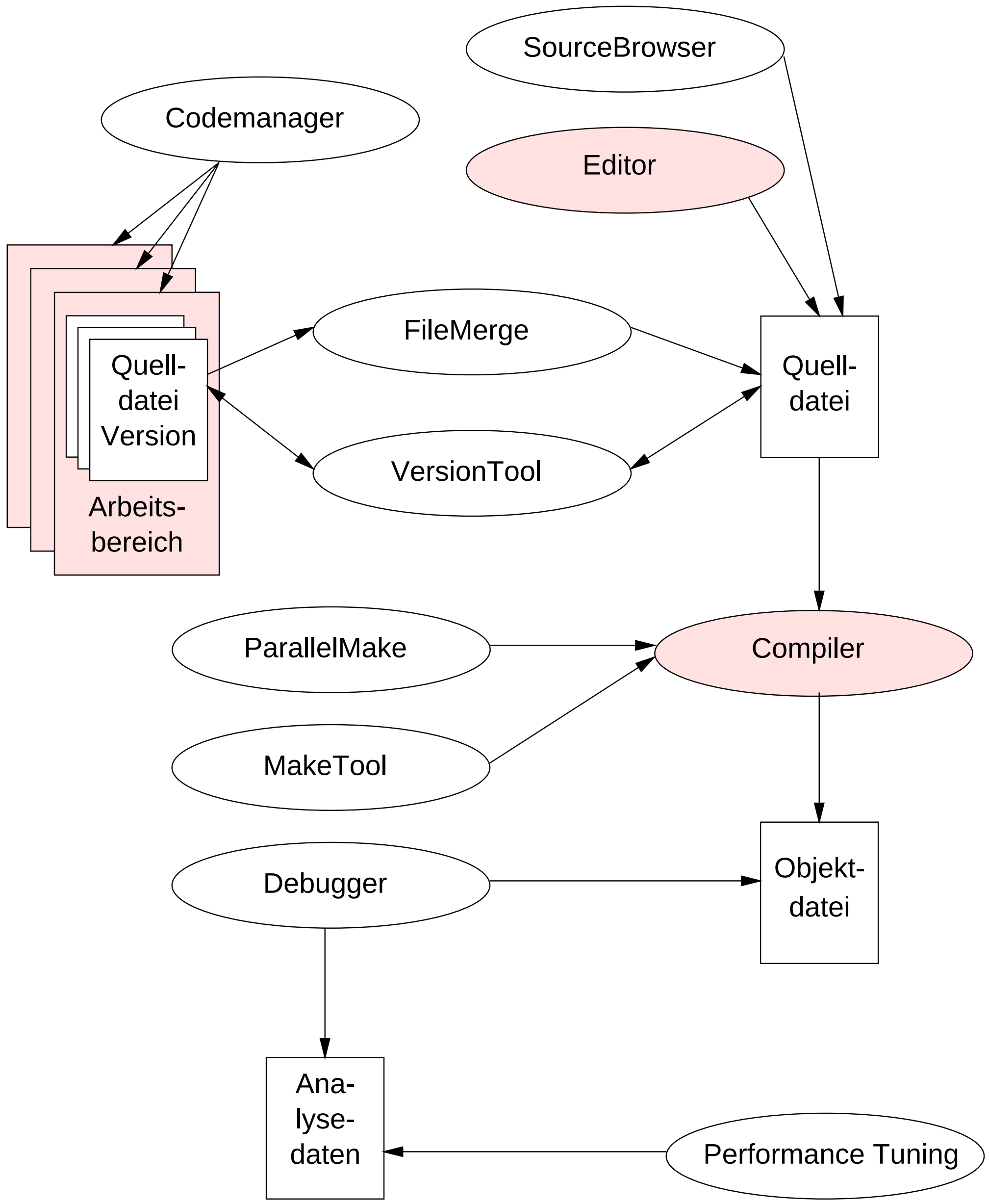


Abbildung 2.2: Überblick über die SPARCWorks-Entwicklungsumgebung.

VersionTool: Bei der Softwareentwicklung insbesondere in einem Team werden die Änderungen an einem Quelltext üblicherweise durch eine Abfolge von Versionen festgehalten. Ein Programm, das diesen Prozeß unterstützt, ist z.B. SCCS [Rochkind 1975]. Es ist Teil der hier beschriebenen Entwicklungsumgebung. SCCS wird über eine Reihe von Befehlen gesteuert, die auf der Kommandozeile durch Angabe entsprechender Optionen in ihrem Verhalten beeinflußt werden können. Um die Bedienung möglichst einfach zu gestalten, kann der Programmierer auf die Funktionalität dieser Versionsverwaltung über ein grafisch orientiertes Werkzeug zugreifen.

FileMerge: Bei der Erstellung mehrerer Entwicklungsstränge für eine Quelldatei und insbesondere bei der Entwicklung in einem Team tritt der Fall auf, daß diese Stränge wieder zusammengeführt werden müssen. Bei diesem Vorgang werden zwei unterschiedliche (aber meist von einer Datei abstammende) Quelldateien zu einer einzigen verschmolzen. Die Aufgabe von FileMerge ist es, dies in einer grafisch unterstützten Weise zu erledigen.

Codemanager: Der Codemanager ist die Komponente, die die Aktivitäten der an einem Projekt beteiligten Entwickler koordiniert. Es wird dazu das Modell des Arbeitsbereichs (*workspace*) eingeführt, welches jedem Entwickler seinen eigenen Entwicklungsbereich zuordnet. Änderungen werden in einem übergeordneten Arbeitsbereich konsolidiert. Dieses Werkzeug macht sich die Versionsverwaltung SCCS und eine Verzeichnisstruktur, die die Organisation in Arbeitsbereiche widerspiegelt, zunutze, um die beschriebenen Aufgaben zu erfüllen.

ParallelMake: Wie das weiter oben beschriebene MakeTool bietet auch das ParallelMake ein grafisches Frontend zum *make*-Programm. Es benutzt jedoch eine spezielle Version, die unabhängig ablaufbare Aktionen in verschiedene Betriebssystemprozesse plaziert und somit die Möglichkeit schafft, diese parallel ausführen zu lassen.

2.2.1 Teamunterstützung

Der SPARCWorks Umgebung liegt ein bestimmtes Modell zugrunde, nach dem die Entwicklung von Software abläuft. Dieses Modell, welches besonders auf die Unterstützung der Softwareentwicklung im Team abzielt, wird in diesem Abschnitt vorgestellt.

Die Unterstützung mehrerer Entwickler, die simultan an einem Projekt arbeiten, wird in der SPARCWorks Umgebung mit dem Konzept der Arbeitsbereiche (*workspace*) realisiert. Jeder Programmierer arbeitet in einem lokalen Bereich mit Kopien der Entwicklungsobjekte, die er zu modifizieren beabsichtigt. In einem Integrationsbereich werden die Änderungen verschiedener lokaler Arbeitsbereiche synchronisiert. Abbildung 2.3 zeigt ein einfaches Modell von Arbeitsbereichen, die in einer Eltern-Kind-Beziehung stehen. Entlang der Pfeile erfolgt, wie später erläutert wird, die Synchronisation in beiden Richtungen. Mehrstufige Modelle sind möglich, in denen Arbeitsbereiche auf mittleren Stufen als Integrationsbereich für unterliegende Bereiche fungieren.

Entwicklungsmodell

Dem Entwicklungsmodell des CodeManagers liegt als Basis der Arbeitsbereich zugrunde. Hierbei handelt es sich um ein Verzeichnis inklusive seiner Unterverzeichnisse, die die entsprechenden Dateien beherbergen. Konzeptuell formen diese Arbeitsbereiche eine Hierarchie, wie sie in Abbildung 2.3 gezeigt ist. Die Entwicklung erfolgt in einem aus drei Phasen bestehenden Zyklus: Aktualisieren, lokal modifizieren und hochschreiben (*copy–modify–merge*).

- Im ersten Schritt werden die Dateien, die modifiziert werden sollen, vom übergeordneten Arbeitsbereich kopiert. Dies kann von mehreren Programmierern gleichzeitig vorgenom-

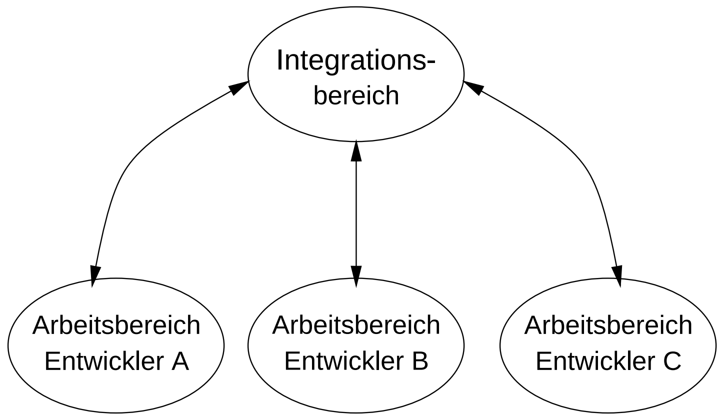


Abbildung 2.3: Organisation von Arbeitsbereichen.

men werden.

- Der Entwickler nimmt in der zweiten Phase Änderungen an den kopierten Dateien vor.
- Die geänderten Dateien müssen nun wieder mit dem übergeordneten Arbeitsbereich, dem Integrationsbereich, zusammengeführt werden. Sie werden dazu lediglich in den Integrationsbereich kopiert.

Der letzte Schritt kann nur erfolgen, wenn keine Änderungen aus einem anderen Arbeitsbereich bereits in den Integrationsbereich eingeflossen sind. Diese würden durch die neuen Änderungen überdeckt werden. In diesem Fall müssen die Änderungen im Integrationsbereich erst im Arbeitsbereich nachvollzogen werden. Erst dann kann der Arbeitsbereich mit dem Integrationsbereich synchronisiert werden. Ein aufgrund der nicht automatischen Durchführbarkeit aufwendiges Zusammenführen von Dateien erfolgt stets im untergeordneten Arbeitsbereich.

Bei der Schilderung des Modells wurde von einer einstufigen Hierarchie zwischen Arbeitsbereichen ausgegangen. Die Erweiterung auf mehrere Stufen erhöht die Komplexität jedoch nur unwesentlich, da ein Austausch von Dateien nur zwischen Arbeitsbereichen angrenzender Ebenen vorgenommen werden kann.

Aus dem geschilderten Modell ergeben sich vier unterschiedliche Beziehungen, in denen ein Eltern-Arbeitsbereich zu einem Kind-Arbeitsbereich stehen kann, die jeweils unterschiedliche Aktionen zur Folge haben können:

1. Keine Aktionen sind notwendig, wenn sowohl Eltern- als auch Kind-Arbeitsbereich unverändert sind.
2. Wurde nur der Kind-Arbeitsbereich geändert, müssen die korrespondierenden Objekte im übergeordneten Arbeitsbereich aktualisiert werden.
3. Wurde nur der Eltern-Arbeitsbereich verändert, so muß dies dem Kind-Arbeitsbereich „mitgeteilt “ werden. Bevor eine Änderung des Kind-Arbeitsbereichs hochgeschrieben werden kann, muß dieser mit dem übergeordneten Bereich synchronisiert werden, was sich in diesem Fall auf das Kopieren der entsprechenden Dateien beschränkt.

4. Wie im vorigen Fall muß auch verfahren werden, wenn beide beteiligten Arbeitsbereiche geändert wurden. In diesem Fall kann es jedoch bei der Synchronisation des Kind-Arbeitsbereiches zu Konflikten kommen, wenn korrespondierende Dateien in beiden Arbeitsbereichen Änderungen erfahren haben. Diese Konflikte müssen manuell aufgelöst werden.

Zusammenfassend läßt sich sagen, daß Änderungen stets im untergeordneten Arbeitsbereich aufgelöst werden, um den übergeordneten konsistent (für alle anderen untergeordneten Arbeitsbereiche) zu halten.

2.2.2 Integration und Basisdienste

Exemplarisch wird in diesem Abschnitt eines der in Abschnitt 2.2 aufgeführten Werkzeuge (VersionTool) dargestellt. Dabei wird der Fokus der Betrachtung auf die Nutzung von Basisdiensten gelegt, bzw. auf die zur Verfügung gestellten Basisdienste.

Bei dem Werkzeug zur Versionsverwaltung handelt es sich um ein grafisches Frontend zum Versionsverwaltungssystem SCCS. SCCS tritt hier in der Rolle des Basisdienstes Versionierungsdienst der Gruppe der Objektverwaltungsdienste auf. Die Funktionalität, die das grafische Frontend hinzufügt, bezieht sich auf die Integration mit den anderen Werkzeugen. Die Integration wird dabei auf den folgenden Ebenen erreicht[2]:

- Datenintegration wird dadurch erreicht, daß alle Werkzeuge auf die Dateien des Dateisystems Zugriff haben. Das Dateisystem übernimmt die Rolle des Objektverwaltungsdienstes für die grundlegenden Funktionen.

- Präsentationsintegration wird erreicht durch die Verwendung einer einheitlichen Benutzeroberfläche mit der Möglichkeit, Daten bzw. Dateien durch Verwendung der Maus zwischen Werkzeugen auszutauschen. Die Funktionalität des GUI-Toolkits wird hier zusätzlich als Kommunikationsdienst genutzt.

- Kontrollflußintegration wird durch die Werkzeuge insofern geboten, als daß sie als eigenständige Programme und Prozesse in ihrer Ablaufreihenfolge nur sehr geringen Einschränkungen unterliegen und in beliebiger Form sich gegenseitig aufrufen können. So können Dateien mit dem SourceBrowser untersucht werden unabhängig davon, ob und wann sie von der Versionsverwaltung in einer schreibfähigen Version bereitgestellt werden. Dies gilt in gleicher Weise allerdings nicht für den Editor.

Eine Unterstützung für fest definierte Prozesse wird von der SPARCWorks Umgebung nicht geboten. Zwar ergeben sich bestimmte Abläufe durch den Entwicklungszyklus eines Programms.

[2] Die Dimension der Rahmenwerkintegration konnte nicht untersucht werden, da sich diese z.B. auf einheitliche Installationsprozeduren u.ä. bezieht. Die SPARCWorks Entwicklungsumgebung wurde jedoch in einem installierten Zustand vorgefunden. Auch ist zu diesem Punkt keine Dokumentation verfügbar. Es kann jedoch davon ausgegangen werden, daß auch diese Dimension der Integration erfüllt wird, da es sich um Werkzeuge eines Herstellers handelt, die zusammen in einem Softwarepaket vertrieben werden.

Dieser Prozeß kann jedoch in keiner Weise festgeschrieben oder in seinem Ablauf erzwungen werden.

Die von den Werkzeugen der SPARCWorks Umgebung genutzten Basisdienste werden in Form von Bibliotheks- und Systemaufrufen bereitgestellt. Zum einen stützen sie sich auf die vom Betriebssystem bereitgestellten Funktionen, wie z.B. zum Erzeugen eines neuen Prozesses. Zum anderen werden Dienste genutzt, die in Bibliotheken zusammengefaßt sind, wie z.B. das Einlesen der in einem Verzeichnis enthaltenen Namen. Durch diese Funktionen werden im wesentlichen die Basisdienste der Gruppen Objektverwaltung und Betriebssystem abgedeckt. Der Bereich der Kommunikationsdienste wird einerseits ebenfalls durch bereits erwähnte Funktionen abgedeckt, es kommen jedoch die ebenfalls als Bibliotheksfunktionen realisierten Dienste des GUI-Toolkits hinzu.

2.3 ABAP/4 Development Workbench

Die ABAP/4 Development Workbench ist die im SAP R/3 System[3] enthaltene Entwicklungsumgebung. Im Gegensatz zu der in Abschnitt 2.2 vorgestellten Umgebung wird sie exemplarisch für eine Entwicklungsumgebung vorgestellt, die die nachstehend aufgeführten zwei Eigenschaften aufweist:

- Die Entwicklungsumgebung ist objektbasiert, d.h. ihre Softwareeinheiten werden in einem Objektspeicher (Repository) abgelegt und sind über eine entsprechende Schnittstelle zugreifbar. Zwischen den Objekten können (je nach Mächtigkeit des zugrundeliegenden Datenmodells) Beziehungen und Invarianten (z.B. Typen) definiert werden.

- Die Entwicklungsumgebung ist sprachgebunden. Fest in die Umgebung integriert ist eine Programmiersprache. Bei diesem Modell einer Entwicklungsumgebung können die Werkzeuge den speziellen Bedürfnissen der Sprache und dem durch sie möglicherweise implizierten Entwicklungsmodell (z.B. objektorientierter Entwurf bei einer objektorientierten Programmiersprache) angepaßt werden.

Die Development Workbench zeichnet sich durch eine hohe Integration in ein anderes System, das SAP R/3 System, aus. Durch die starke Verflechtung der Programmiersprache ABAP/4[4] mit den Werkzeugen der Entwicklungsumgebung ist es sinnvoll, auch Aspekte, die eigentlich der Sprache selbst zugeordnet werden müssen, hier zu behandeln. Dazu gehören die zahlreichen Strukturierungsmöglichkeiten. Wie in [Will et al. 1995] nachzulesen ist, liegt u.a. in der Strukturierbarkeit eine der Stärken von ABAP/4, obwohl die Strukturierung nicht ausschließlich durch sprachliche Mittel erreicht wird. Abbildung 2.4 zeigt die enge Verzahnung der Komponenten der ABAP/4 Entwicklungsumgebung. Zu beachten ist, daß keine direkte Korrespondenz zwischen Werkzeugen der Umgebung und Gestaltungsblöcken der Abbildung besteht.

[3] Die im Rahmen dieser Arbeit vorgenommenen Untersuchungen basieren auf der Version 2.2D des R/3 Systems.

[4] ABAP/4: „Advanced Business Application Programming“. Die Zahl vier im Namen unterstreicht die Zugehörigkeit zur Gruppe der Programmiersprachen der vierten Generation.

Es werden die folgenden Aspekte untersucht: Abschnitt 2.3.1 beleuchtet die Möglichkeiten, die die Programmiersprache ABAP/4 und die Development Workbench zur Strukturierung bereitstellen. Abschnitt 2.3.2 befaßt sich mit Versionierung. Das Korrektur- und Transportwesen regelt die Unterstützung für die Entwicklung in einer Gruppe mehrerer Entwickler und ist für den Austausch von Entwicklungsobjekten[5] zwischen unterschiedlichen Systemen verantwortlich. Es wird in Abschnitt 2.3.3 vorgestellt. Eine Bewertung beschließt die Ausführungen zur ABAP/4 Entwicklungsumgebung.

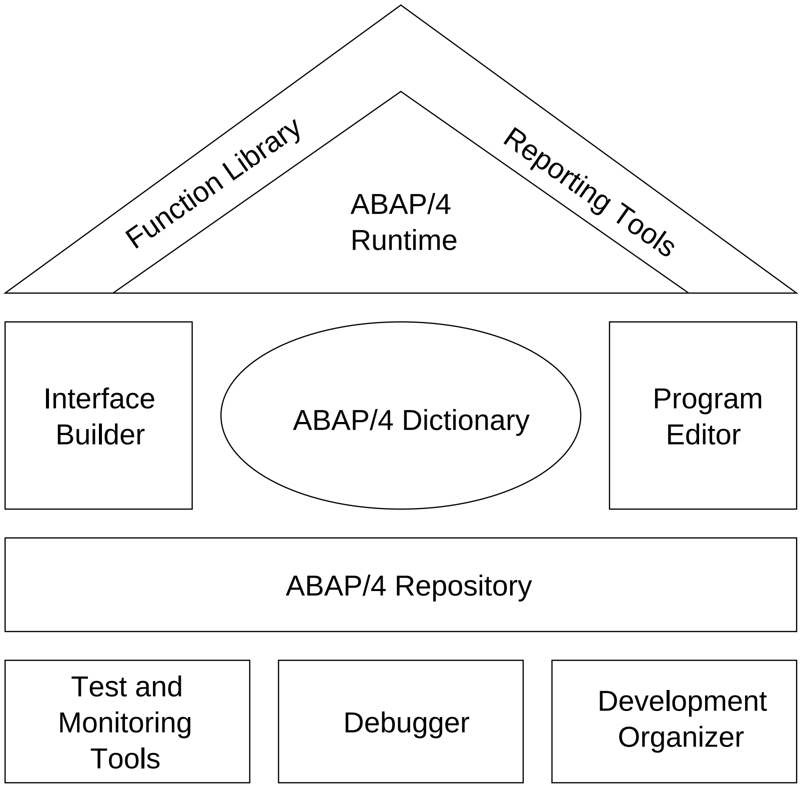


Abbildung 2.4: Architektur der ABAP/4-Entwicklungsumgebung (aus [SAP 1994])

Aspekte des Konfigurationsmanagements und weitere Unterstützung für ein Lebenszyklusmodell (*life cycle support*) sind nicht in der Entwicklungsumgebung berücksichtigt [III 1995]. Nicht in die Betrachtung einbezogen wird außerdem die Systemgenerierung. ABAP/4 ist eine interpretative Sprache, deren Zwischencode zur Laufzeit abgearbeitet wird. Der Zwischencode wird beim ersten Aufruf des Programms erzeugt und anschließend persistent in Datenbanktabellen abgelegt. Er steht bei weiteren Programmaufrufen wieder zur Verfügung. Die Generierung dieses Codes erfolgt vollständig transparent für den Anwender und Programmierer, obwohl letzterer die Zwischencodeerzeugung auch explizit starten kann. Abhängigkeiten zwischen verschie-

[5]Als Entwicklungsobjekt wird in diesem Kontext alles aufgefaßt, was mit der Entwicklung von Programmen zu tun hat. Dies reicht von Quelltexten, Bildschirmmasken, Tabellendefinitionen bis hin zu Dokumentationen und Nachrichten.

denen Programmteilen und dem Repository[6] werden zur Laufzeit aufgelöst.

Die ABAP/4 Development Workbench unterstützt schwerpunktmäßig die Entwicklung betriebswirtschaftlicher Anwendungen mit grafischen Oberflächen. Dadurch wird dem Programmierer eine bestimmte Art der Programmgestaltung bereits vorgegeben, die in der Entwicklung von Bildschirmmasken mit entsprechender Verarbeitungslogik besteht. Eine ABAP/4-Anwendung kann über eine Abfolge dieser Bildschirmmasken, Dynpros[7], beschrieben werden bzw. ist immer eine Abfolge eines oder mehrerer Dynpros. Durch Benutzung von ABAP/4 und der Development Workbench ist der Programmierer somit auf einen engen Ausschnitt des möglichen Anwendungsspektrums beschränkt, er hat es nicht mit einer Sprache für beliebige Anwendungen (*general purpose language*) zu tun. Die Folge sind Restriktionen, wie sie von anderen Sprachen nicht bekannt sind. Sie werden in den nächsten Unterabschnitten genauer erläutert.

2.3.1 Strukturierung

Wie Modula-2 oder Ada bietet auch ABAP/4 eine Reihe von Strukturierungsmechanismen, wie Funktionen, Moduln usw. Alle Strukturierungselemente werden hier der Reihe nach behandelt. Leider ist eine klare Begrifflichkeit nicht an allen Stellen möglich, da sich auch die von der SAP verfügbare Dokumentation in einigen Punkten widerspricht.

Bereits an dieser Stelle wird der Begriff der Entwicklungsklassen eingeführt. Im Abschnitt 2.3.3 wird er in anderem Zusammenhang wieder aufgegriffen. Jedes Entwicklungsobjekt ist einer Entwicklungsklasse zugeordnet. Im SAP R/3 System sind eine Reihe von Entwicklungsklassen vordefiniert. Für die Neudefinition von Klassen stehen dem Programmierer durch die begrenzte Namenslänge (zwei Zeichen) und die Einschränkung auf bestimmte Anfangsbuchstaben (Y und Z) nur eine geringe Zahl von möglichen Namen zur Verfügung. Als Strukturierungskonzept für große Entwicklungen scheiden Entwicklungsklassen daher aus, auch weil sie keine neuen Namensräume festlegen. Jedes Entwicklungsobjekt einer bestimmten Kategorie, z.B. eine Tabelle oder ein Programm, muß über alle Entwicklungsklassen hinweg einen eindeutigen Namen besitzen.

Unterprogramme

Unterprogramme entsprechen Funktionen, wie sie bspw. aus C oder Pascal bekannt sind, mit dem Unterschied, daß sie sowohl mehrere Eingabe- als auch mehrere Ausgabeparameter besitzen können. Sie werden innerhalb eines Programmes auf syntaktischer Ebene definiert und sind damit keine eigenständigen Entwicklungsobjekte.

Ebenfalls sind Unterprogramme keine Objekte erster Klasse. Sie können also weder als Parameter an andere Unterprogramme übergeben, noch können Variablen, die Unterprogramme enthalten, mit entsprechenden Operationen angelegt werden. Auch eine Schachtelung ist nicht möglich.

[6]Das Repository ist die zentrale Ablagekomponente, die alle Entwicklungsobjekte in den Tabellen des unterliegenden relationalen Datenbanksystems speichert. Insbesondere ist hier das Data Dictionary enthalten, welches die Metainformation der Tabellen beinhaltet.

[7]Dynpro: „Dynamisches Programm“.

Innerhalb von Unterroutinen ist die Verwendung von lokalen Variablen möglich. Diese überdecken dabei die möglicherweise bereits global definierten Variablen. Zu undefiniertem Verhalten kommt es, wenn eine lokale Variable in Konflikt zu einem gleichnamigen Funktionsargument steht [Matzke 1996]. Der Syntaxüberprüfer der ABAP/4 Development Workbench erkennt diesen unerwünschten Effekt nicht.

Dynpro

Wie bereits erwähnt, erfolgt die Programmierung einer Anwendung in ABAP/4 in Dynpros, die zu Dynproketten verknüpft werden. Dynpros nehmen damit eine zentrale Rolle in der Entwicklung ein, da sie den Programmfluß auf oberster Ebene vorgeben. Ein Dynpro besteht aus einer Bildschirmmaske und zwei[8] Codeteilen, auch Ereignisse[9] genannt, die jeweils vor Anzeige der Maske bzw. nach Beendigung der Bearbeitung durch den Benutzer abgearbeitet werden. Bei den Anweisungen innerhalb der Verarbeitungslogik handelt es sich nicht um herkömmliche ABAP/4 Anweisungen. Vielmehr kann hier nur ein Teil des Sprachvorrats benutzt werden. Üblicherweise erfolgen an dieser Stelle Aufrufe an Moduln, die ihrerseits im Modulpool formuliert sind bzw. wird ein Teil der Eingabeprüfungen vorgenommen.

Das Besondere an den Moduln ist, daß sie im Gegensatz zu den beschriebenen Unterprogrammen nur aus der Ablauflogik heraus aufgerufen werden können. Es ist nicht möglich, Moduln zu definieren und diese aus einem Programm (Include, Funktionsbaustein oder Modulpool) anzusprechen. Eine weitere Einschränkung stellt die Parameterübergabe dar, die es nicht gestattet, Argumente zu übergeben. Ein Modul in ABAP/4 ist demnach nicht mit der Semantik eines Moduls vergleichbar, wie sie z.B. in Modula-2 Anwendung findet. Vielmehr entspricht der Modul einer Funktion, die darauf angewiesen ist, auf globalen Variablen des Programms zu arbeiten und in bestimmten Programmsituationen aufgerufen zu werden.

Programm

Der Begriff des Programms wird von SAP in unterschiedlicher Weise benutzt. Er bezeichnet eine ablaufbare Einheit und den zugehörigen Quelltext, andererseits ist *Programm* ein Oberbegriff für Programmtypen, von denen es die folgenden gibt:

- Online-Report
- Include-Report
- Funktionsbaustein
- Modulpool

[8] Zwingend erforderlich sind die Ereignisse PBO (*process before output*) und PAI (*process after input*), die der Dynproprozessor vor bzw. nach Anzeige der Bildschirmmaske bearbeitet. Zusätzlich dienen die Ereignisse POH (*process on help request*) und POV (*process on value request*) dazu, Hilfedialoge aufzubauen.

[9] SAP bezeichnet ABAP/4 auch als ereignisgesteuert. Hierbei handelt es sich jedoch nicht um ein bspw. dem X Window System vergleichbares System, welches eine Vielzahl von Ereignissen verschiedenster Art behandelt. Vielmehr beschränkt sich ABAP/4 bei der Bearbeitung von Dynpros auf die vier beschriebenen Ereignisse.

Der Online-Report ist mit einem Programm vergleichbar, wie es in anderen Sprachen existiert. Zwar weist er starke Ähnlichkeit zum weiter unten beschriebenen Modulpool auf, im Gegensatz zu diesem dient er in der Regel jedoch der Entwicklung listengenerierender Programme. Dieser Verwendungszweck wird durch die Development Workbench jedoch in keiner Weise vorgegeben.

Include

Include-Programme dienen dazu, größere Programme in kleinere Einheiten zu zerlegen. Sie enthalten im allgemeinen keinen eigenständig ablaufbaren Code, sondern werden über eine entsprechende Anweisung in ein Programm eingefügt. Dieses Einfügen ist mit der C-Präprozessoranweisung `#include` vergleichbar.

Da es sich hierbei um eine rein textuelle Ersetzung der Einfügeanweisung mit dem korrespondierenden Text handelt, werden keine neuen Namensräume erzeugt. Vielmehr befinden sich alle Bezeichner eines Programms auf einer Ebene, auch wenn die Einfügeanweisungen geschachtelt sind.

Der durch Include-Programme erreichbare Nutzen besteht in einer besseren Übersicht innerhalb eines Programms. Ohne eine einhergehende Schachtelungsmöglichkeit von Namensräumen kann es jedoch zu Namenskonflikten kommen, die aufgrund der unterschiedlichen Definitionsorte der Namen schwer zu durchschauen sind. Insbesondere scheint auch die in [AG] vorgeschlagene Auslagerung einzelner Unterprogramme in sogenannte Unterroutinenpools (Include-Programme) aus diesem Grund nicht sinnvoll zu sein.

Um das auch von der SAP erkannte Problem der Namenskonflikte zu lösen, muß der Programmierer strenge Namenskonventionen einhalten. Der Name einer Sammlung von Unterroutinen setzte sich z.B. aus den Buchstaben `SAPF` gefolgt von einem Buchstaben und zwei Ziffern für den eigentlichen Namen gefolgt von einem Typkennzeichen zusammen. Z.B.:

```
SAPFM03S
```

Dies steht für die Unterroutine *M03* mit dem Typ *S*. Include-Programme werden nach dem Muster

```
FaxxyE01
```

benannt, wobei `axxy` für einen Namen steht, wie er oben beschrieben ist. `E01` bezeichnet ein Kürzel, welches jeder in diesem Include-Programm vorhandenen Unterroutine vorangestellt wird. In diesem Fall handelt es sich um extern aufrufbare Unterroutinen, interne werden z.B. mit `F01` bezeichnet.

Funktionsbaustein

Funktionsbausteine besitzen eine ähnliche Semantik wie Unterprogramme mit dem Unterschied, daß sie eine klar definierte Schnittstelle haben und nicht auf globale Variablen des aufrufenden Programms zugreifen können. Der Zugriff erfolgt ausschließlich über ihre Schnittstelle. Alle Funktionsbausteine eines Systems sind in einer zentralen Funktionsbibliothek abgelegt. Eine

Schachtelung ist nicht vorgesehen, was die Möglichkeiten zur Benennung stark einschränkt, zumal für den Namen lediglich 30 Zeichen zur Verfügung stehen.

Funktionsbausteine werden zu Funktionsgruppen zusammengefaßt. Innerhalb dieser Gruppe können die Bausteine auf gemeinsame Daten sowie auf dort definierte Unterprogramme zugreifen. Beide sind lokal und nach außen nicht sichtbar. Eine Funktionsgruppe entspricht im Sinne seines Typs einem Programm, da das sie einleitende Schlüsselwort (*FUNCTION-POOL*) äquivalent zu dem eines Programms ist (*REPORT* bzw. *PROGRAM*). Eine Funktionsgruppe ist jedoch nicht eigenständig ablauffähig. Sie definiert bzgl. der Funktionsbausteinbezeichner keine neuen Sichtbarkeitsbereiche.

Da pro System jeweils nur eine Funktionsbibliothek existiert, kann der Verwendungszweck nur auf allgemeingültige Aufgaben beschränkt sein. Die Funktionsbibliothek gleicht damit der aus C bekannten Standardbibliothek (*libc*), die eine bloße Ansammlung von allgemein nützlichen Funktionen darstellt. Eine Programmstrukturierung läßt sich mit Funktionsbausteinen nicht bewerkstelligen, da die Benennungs- und Schachtelungsmöglichkeiten zu eingeschränkt sind und auch immer nur eine Bibliothek existiert.

Modulpool

Ein Modulpool nimmt den Quelltext der Moduln auf, die aus der Ablauflogik eines Dynpros heraus aufgerufen werden. Zwischen Modulpool und Dynpros besteht eine sehr enge Beziehung. Ein Dynpro muß immer genau einem Modulpool zugeordnet sein, in dem sich die Moduln der Ablauflogik befinden.

Der Hauptzweck eines Modulpools ist organisatorischer Art. Er dient als Zuordnungspunkt für andere Elemente einer Anwendung, wie z.B. Dynpros oder Oberflächen[10]. Auch der Transaktionscode, der zum Aufruf eines Programms dient, wird am Modulpool und einem Dynpro festgemacht.

Modulpools stehen im gleichen Namensraum wie die anderen Entwicklungsobjekte vom Typ *Programm*: Funktionsbaustein, Include-Programm, Online-Programm. Dies gilt auch für Programme in unterschiedlichen Entwicklungsklassen. Diese bieten nur eine organisatorische Unterteilung und keine Sichtbarkeitsbereiche.

Dialogbaustein

Dialogbausteine sind ein weiteres Strukturierungskonzept, welches in einer höheren Abstraktionsebene angesiedelt ist als das der Funktionsbausteine. Zwar bieten sie ebenfalls eine fest definierte Aufrufschnittstelle, vereinen jedoch neben Unterroutinen auch Dynpros und Oberflächen in ihrem Modulpool.

[10]Mit Oberflächen bezeichnet man die Gesamtheit von Menüs, Druck- und Symboltasten, die zu einer Bildschirmmaske angezeigt werden können. Zur näheren Begriffsklärung und weiteren Erläuterungen siehe [Koch 1996a].

2.3.2 Versionierung

Von jedem Entwicklungsobjekt können Versionen angelegt werden. Der Begriff der Version wird von der SAP jedoch in Abwandlung der üblichen Bedeutung verwendet. Um dies zu verdeutlichen, wird kurz die generelle Arbeitsweise des Repositorys erläutert, in dem alle Entwicklungsobjekte abgelegt sind (s.a. Abbildung 2.5).

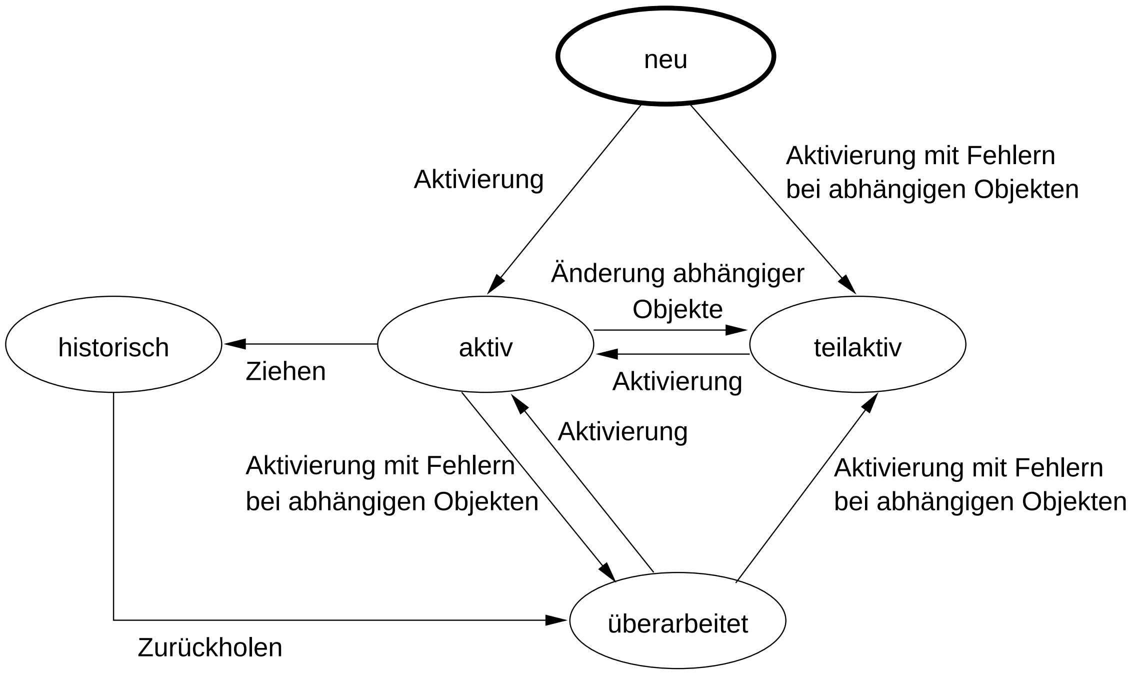


Abbildung 2.5: Zustände von ABAP/4 Entwicklungsobjekten.

Ein Entwicklungsobjekt muß nach dem Anlegen (Zustand *neu*) zunächst aktiviert werden, bevor es von anderen Entwicklungsobjekten referenziert werden kann. Beide Zustände, wie auch alle anderen in Abbildung 2.5 aufgeführten, werden als Version eines Objektes bezeichnet. Tritt beim Aktivieren ein Fehler bei abhängigen Objekten auf, weil diese z.B. syntaktisch nicht korrekt sind, so bezeichnet man das Objekt als teilaktiv.

Die Abfolge unterschiedlicher Entwicklungsstände wird hauptsächlich mit den historischen Versionen dokumentiert, während der Entwicklungsprozeß durch eine Zahl weiterer Versionen modelliert wird. Die Versionsabfolge der historischen Zustände ist in diesem Modell stets eine sequentielle Abfolge, da eine zurückgeholte, historische Version die überarbeitete überschreibt. Eine Verzweigung des Versionsabfolgegraphen ist nur mit großen Einschränkungen, nämlich durch eigene Sequentialisierung des Graphen, machbar. Das Zusammenführen zweier unterschiedlicher Entwicklungsstränge erfordert manuellen Editieraufwand, da kein entsprechendes Werkzeug zur Verfügung steht.

2.3.3 Korrektur- und Transportwesen

Das Korrekur- und Transportwesen (KTW) ist ein wesentlicher Teil der ABAP/4 Development Workbench, der besonders die Entwicklung im Team unterstützt. Es besteht, wie der Name bereits andeutet, aus zwei Teilen:

- Das Korrekturkontrollsystem dient dazu, Entwicklungen unterschiedlicher Programmierer in unterschiedlichen Systemen zu koordinieren. Es verfolgt dabei eine äußerst restriktive Methode, indem es, sobald an einem Entwicklungsobjekt gearbeitet wird, jedem anderen Entwickler einen ändernden Zugriff darauf verweigert, wenn keine entsprechenden Maßnahmen zur Freigabe getroffen werden.

- Da eine R/3 Installation meist aus mehreren parallelen Systemen besteht, ist es notwendig, in einem System getätigte Änderungen auch in ein anderes System übernehmen zu können. Das ist die Aufgabe des Transportwesens.

An dieser Stelle können einerseits aus Platzgründen nur grundsätzliche Abläufe und Konzepte vorgestellt werden, andererseits kann das KTW in seinem Verhalten durch Anpassungen (*customizing*) stark verändert werden, so daß sich eine detaillierte Beschreibung immer auf eine konkrete Installation beziehen muß.

Systeme

Der schon mehrfach verwendete Begriff des Systems soll an dieser Stelle einer genaueren Betrachtung unterzogen werden, mit dem Ziel, eine möglichst exakte Definition zu erlangen.

Ein SAP R/3 System beinhaltet die komplette Entwicklungsumgebung, je nach Installation eine Reihe von Anwendungen[11] und Anwenderdaten. Da es nur ein Exemplar jeder Softwarekomponente gibt, würden sich Änderungen an diesen direkt auf die Anwendungen auswirken. Es ist also notwendig, für den Programmierer eine Möglichkeit bereitzustellen, neue Programme zu testen, ohne die bestehenden Anwendungen zu stören. Hierzu werden von der SAP verschiedene System-Begriffe eingeführt [Will et al. 1995]. Abbildung 2.6 zeigt die durch die Systeme gebildete Hierarchie und die erlaubten Transportrichtungen.

Entwicklungssystem: Dieses System ist, wie der Name zum Ausdruck bringt, für die Entwicklung vorgesehen. Allerdings werden hier nur solche Komponenten entwickelt, die besonders kritisch sind und deshalb von der übrigen Entwicklung im Integrationssystem getrennt werden müssen.

Integrationssystem: Das ist das eigentliche Entwicklungssystem, in dem die Programmierung normaler Anwendungen und deren Tests erfolgt.

[11] Da auf den Anwendungsaspekt des SAP R/3 Systems nicht näher eingegangen wird, wird vereinfachend von Anwendungen gesprochen. Diese setzen sich aus Hauptanwendungen und Hauptkomponenten zusammen und werden oft auch als Module bezeichnet. Nähere Informationen finden sich in [Koch 1996a].

Konsolidierungssystem: In diesem System werden die unterschiedlichen Entwicklungen der Entwicklungs- und Integrationssysteme zusammengeführt, wobei ein direkter Transport von Objekten aus einem Entwicklungssystem in ein Konsolidierungssystem nicht erlaubt ist.

Auslieferungssystem: Hierher werden die Entwicklungen aus dem Konsolidierungssystem letztendlich transportiert. Beim Auslieferungssystem kann es sich um ein produktiv eingesetztes System handeln.

Es müssen nicht immer alle Systeme installiert werden. Meist kommen aber Integrations- und Konsolidierungssystem zum Einsatz. Werden nicht alle Systemtypen verwendet, kann auch das Konsolidierungssystem das Produktivsystem sein.

Der Sinn dieser verschiedenen Systeme besteht darin, die Entwicklungsarbeit des Programmierers von der Arbeit des Anwenders zu entkoppeln und somit zu verhindern, daß produktiv eingesetzte Daten oder Programme geändert werden. Dies könnte allerdings auch mit einer entsprechend leistungsfähigen Versionskontrolle, die die Verzweigung des Versionsabfolgegraphens erlaubt, in Kombination z.B. mit dem Arbeitsbereichs-Modell der Teamwareumgebung erreicht werden.

Der System-Begriff ist eine Unterteilung auf der Ebene der Entwicklungsklassen. Eine Instanz eines SAP R/3 Systems kann somit für eine Klasse Konsolidierungs- und für eine andere Klasse Integrationssystem sein.

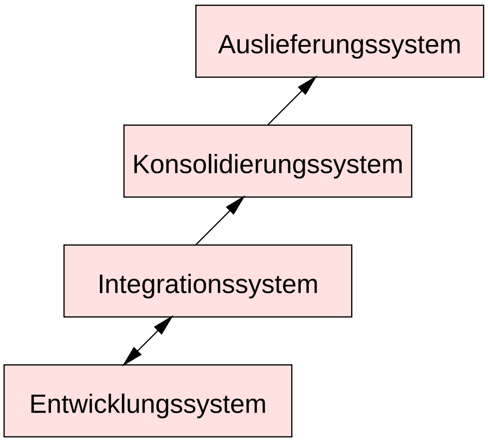


Abbildung 2.6: Arten von Systemen im R/3 System und die erlaubten Transportwege zwischen ihnen (aus [Will et al. 1995]).

Korrekturkontrollsystem

Kern dieses Teils des KTW ist das Konzept des Originalobjektes. Ein Entwicklungsobjekt kann danach jeweils nur im Original verändert werden. Da ein Objekt nur in einem der installierten Systeme im Original vorliegt, kann es auch nur dort bearbeitet werden. Dies hat zur Folge, daß jeweils nur ein Entwickler bzw. Team an einem Objekt im Originalsystem arbeiten kann, parallele Entwicklung zwischen mehreren Entwicklern in unterschiedlichen Teams bzw. verschiedenen Teams ist damit praktisch ausgeschlossen. Das Objekt kann einem anderen System als Originalsystem zugeordnet werden.

Bei der Erstellung wird einem Objekt ein Entwickler oder ein Team von Entwicklern zugeordnet. Es kann dann auf zwei unterschiedlichen Ebenen für den Zugriff gesperrt werden. Nach Erzeugung des Objektes ist es automatisch auf erster Ebene gesperrt, eine Korrektur auf dieses Objekt wurde eröffnet. Es kann dann von Entwicklern des gleichen Teams ebenfalls modifiziert werden, wobei jeder Entwickler eine eigene Korrektur eröffnet und seine eigene Version erhält. Dieser Vorgang wird als Vernetzen einer Korrektur bezeichnet. Auf der zweiten Stufe können Objekte vollständig gegen Modifikation anderer gesperrt werden.

Eine Ausnahme des Originalkonzeptes bilden die Reparaturen. Sie erlauben es, auch an Kopien eines Objektes, also durch Transporte entstandenen Objektes, Änderungen durchzuführen. Durch Reparatur geänderte Objekte müssen jedoch wieder mit dem Original im Originalsystem zusammengeführt werden, bevor sie durch neue Versionen aktualisiert werden können. Reparaturen eignen sich somit nicht zur Parallelentwicklung.

Transportwesen

Hauptaufgabe des Transportwesens ist es, die Entwicklungsobjekte zwischen den verschiedenen Systemen zu bewegen. Dabei gibt es die in Abbildung 2.6 gezeigten Beschränkungen. Existieren mehrere Systeme einer Art, kann auch zwischen diesen Systemen ein Austausch erfolgen. SAP selbst nutzt das Transportwesen, um Aktualisierungen ihres Systems in die Kundensysteme einzuspielen.

Das Transportwesen arbeitet auf Entwicklungsumgebungsobjekten. Es handelt sich dabei um die Zusammenfassung von Entwicklungsobjekten, die in Beziehungen zueinander stehen. Entwicklungsumgebungsobjekte sind darüberhinaus in Typen eingeteilt, die eine genaue Spezifikation der abhängigen Entwicklungsobjekte erlaubt. Z.B. hat das Entwicklungsumgebungsobjekt `R3TR PROG` den Objekttyp `R3TR`[12]. `R3TR` legt dabei fest, daß ein ABAP/4 Programm und von ihm abhängige Bestandteile gemeint sind. Zusammengenommen umfaßt das Entwicklungsumgebungsobjekt ein ABAP/4 Programm mit seinen Oberflächen, Dynpros, Texten und Dokumentationen. Dieses Entwicklungsumgebungsobjekt kann nun auf ein konkretes Entwicklungsobjekt angewendet werden und legt damit fest, welche Objekte transportiert werden sollen.

Eine Entwicklungsklasse umfaßt in einer weiteren Gliederungsebene mehrere Entwicklungsumgebungsobjekte. Der Transport kann auch auf deren Basis durchgeführt werden.

Der Transport zwischen zwei Systemen erfolgt dann in zwei Phasen:

- In der Exportphase werden die durch Entwicklungsklassen oder Entwicklungsumgebungsobjekte spezifizierten konkreten Entwicklungsobjekte auf Betriebssystemebene des Quellsystems in eine Datei geschrieben. Das Format dieser Datei ist plattformunabhängig.

[12]Die Begriffsbildung in der Literatur ist an dieser Stelle nicht eindeutig. Gelegentlich wird auch `PROG` als Objekttyp und `R3TR` als Programm-ID bezeichnet.

- Die Importphase wird auf dem Zielsystem explizit angestoßen. Hierbei kann es sich um die gleiche oder eine andere SAP R/3 Instanz handeln.

2.3.4 Integration und Basisdienste

Die ABAP/4 Development Workbench zeichnet sich durch Integration auf verschiedenen Ebenen aus:

- Die Development Workbench arbeitet mit einer zentralen Datenbank, dem Repository, das alle Entwicklungsobjekte und Beziehungen zwischen ihnen verwaltet. Alle Komponenten greifen auf diese gemeinsame Datenbasis zu. Auf diese Weise wird Datenintegration erreicht.

- Kontrollflußintegration wird in der Umgebung erreicht, da alle Werkzeuge in beliebiger Reihenfolge ausgeführt werden können.

- Durch die einheitliche Verwendung der durch ABAP/4 bereitgestellten grafischen Elemente zur Gestaltung von Benutzeroberflächen, erhalten alle Komponenten ein einheitliches Erscheinungsbild.

Durch ihre enge Verflechtung sind die einzelnen Werkzeuge der Entwicklungsumgebung als Programm nicht voneinander zu unterscheiden. Die Unterteilung in verschiedene Komponenten erfolgt mehr auf konzeptueller Ebene.

Die Basisdienste, die den Werkzeugen zur Verfügung stehen, werden einerseits durch das Repository als Objektverwaltung bereitgestellt, andererseits durch das SAP R/3 System und seine Programmiersprache ABAP/4, welche grundlegende Betriebssystem- und Kommunikationsfunktionalität liefern bzw. für den Programmierer erledigen.

Kapitel 3

Systemstrukturierung

Der zweite Teil dieser Arbeit untersucht Konzepte zur Systemstrukturierung anhand von Programmiersprachen, die in der Praxis weite Verbreitung gefunden haben oder aufgrund ihrer Strukturierungskonzepte interessant sind. Es werden Sprachen aus unterschiedlichen Sprachklassen genauer untersucht. Modula-2 steht für eine imperative Sprache mit einem Modulkonzept. Dagegen ist C++ ein Vertreter einer objektorientierten Sprache, der ein Klassenkonzept in die Sprache einführt. C++ ist ferner durch die Abstammung von dem imperativen, nicht objektorientierten C beachtenswert, da hierdurch gezeigt werden kann, inwieweit C++ trotz dieser Abstammung geeignetere als die in C realisierten Strukturierungskonzepte verwirklicht, ohne die Kompatibilität zu seinem Vorläufer zu verlieren. Schließlich wird das persistente Programmiersystem Napier88 betrachtet. Hier ist besonders der Vergleich mit den Sprachen TL bzw. TLMIN interessant, da eine Vielzahl der Konzepte wie orthogonale Persistenz, statische strenge Typüberprüfung in beiden Systemen (Napier88 und Tycoon) umgesetzt sind. Eine Diskussion der im Rahmen dieser Arbeit entwickelten Strukturierungskonzepte von TLMIN schließt sich in Kapitel 5 an.

Besonderes Augenmerk wird auf die Untersuchung von Sichtbarkeitsbereichen gerichtet, da sie neben Bindungstechniken eine entscheidende Rolle bei der Programmierung im Großen und damit bei der Systemstrukturierung spielen. Sie geben dem Programmierer die Möglichkeit, Bezeichner explizit sichtbar zu machen oder zu verstecken und somit den geordneten Zugriff zu steuern. Weiterhin bedeutet die Kapselung von Bezeichnern in Modulen und Klassen eine Manipulation von Sichtbarkeitsbereichen. Bestimmte Namen werden beim Vorgang der Kapselung versteckt, während andere sichtbar sind bzw. bleiben. Die Vorteile für diese Art der Programmierung können wie folgt zusammengefaßt werden [Meyer 1988; Koch 1996b]:

Separate Übersetzung: Programme können in kleine überschaubare Einheiten (meist auf Dateiebene) zerlegt werden. Im Vergleich zu monolithischen Systemen, die aus einer großen Programmdatei bestehen und stets komplett übersetzt werden müssen, bietet separate Übersetzung die Möglichkeit, den Übersetzungsaufwand nach einer Änderung gering zu halten. Es müssen nur die Programmteile neu übersetzt werden, die im Sichtbarkeitsbereich der geänderten Bezeichner liegen. Dies ist meist nur eine kleinere Anzahl der zu einem System gehörenden Einheiten, da andere Module, die nicht von der Änderung betroffen sind, unverändert bleiben können, da die geänderten Bezeichner dort nicht sichtbar sind und somit nicht benutzt werden können.

Separate Übersetzung gewährleistet darüber hinaus die Typkonsistenz, wie sie in einem monolithischen System zugesichert werden kann, über Modulgrenzen (Dateigrenzen) hinweg.

Kapselung: Hierunter versteht man das Zusammenfassen z.B. eines Datentyps mit den auf ihn anwendbaren Funktionen in der Art, daß nur über diese Funktionen auf den Zustand des Datentyps zugegriffen werden kann. Die Integrität der gekapselten Daten und die Flexibilität in der zu wählenden Implementierung können so in höchstem Maße gewährleistet werden. Die Manipulation der Sichtbarkeit einzelner Bezeichner eines Moduls erlaubt es, den geordneten Zugriff darauf zu steuern.

Durch Kapselung ist es möglich, von Implementierungsdetails einer Softwareeinheit (Modul, Klassen o.ä.) zu abstrahieren und sich auf das Wesentliche zu konzentrieren (*information hiding*), da nur die Bezeichner sichtbar sind, auf die zugegriffen werden darf und kann. Dies ist eines der wichtigsten Hilfsmittel, um sicher und effizient große Systeme zu entwickeln [Wirth 1979]. Da der Zugriff auf ein Modul nur über die Schnittstelle möglich ist, kann nicht auf die Implementierung Bezug genommen werden, die Implementierung ist sicher gegenüber unerlaubtem Zugriff. Die Schnittstelle eines Moduls kann als ein „black box interface“ angesehen werden, das die Operationen usw. spezifiziert. Sie legt die Menge der sichtbaren Bezeichner fest. Die Implementierung ist vollständig im Implementationsteil enthalten und für den Benutzer nicht zugreifbar. Die Auswirkungen von Änderungen lassen sich lokalisieren und somit die Wartbarkeit erhöhen. Die Lokalität erleichtert gleichzeitig die Wiederverwendung von Modulen, da die Abhängigkeiten zwischen Modulen klar definiert sind und keine versteckten Beziehungen existieren. Das steigert die Effizienz bei der Entwicklung großer Systeme, da bereits existierende Softwarebausteine wiederverwendet werden können.

Hierarchische Strukturierung: Hierarchische Strukturierung unterstützt die Wartungsfreundlichkeit. Während bei einer Programmiersprache wie C[1] alle Prozeduren in einem globalen Sichtbarkeitsbereich definiert sind, kann der Programmierer durch Module strukturieren. Er kann die Sichtbarkeitsbereiche der einzelnen Softwareeinheiten explizit steuern. Dies hat hauptsächlich zwei Vorteile:

- Es kann durch Beschränkung der Menge der sichtbaren Bezeichner verhindert werden, daß verschiedene Prozeduren sich in undisziplinierter Weise gegenseitig benutzen. Stattdessen wird eine Benutzt-Relation zwischen den Modulen automatisch ein gerichteter, (normalerweise) azyklischer Graph[2].

[1] In C sind alle nicht ausdrücklich als lokal zu einer Datei (*static*) definierten Variablen- und Funktionsbezeichner global in der gesamten Übersetzungseinheit sichtbar [Kernighan und Richie 1990]. Das liegt an der impliziten Deklaration von Bezeichnern als extern, also dateiübergreifend.

[2] Sprachen, die ein Modul- oder Klassenkonzept unterstützen, schließen nicht notwendigerweise die Existenz von zyklischen Abhängigkeiten ihrer Strukturierungseinheiten aus. In C sind z.B. zyklische Abhängigkeiten zwischen

- Die Wiederverwendung von Modulen bzw. Modulhierarchien wird erleichtert, da die verwendeten Module durch die Struktur der Abhängigkeiten bekannt sind. Das kommt vor allem dann zum Tragen, wenn Module wiederum in größeren Einheiten wie Bibliotheken zusammengefaßt werden können.

Unabhängige Entwicklung: Die Verbindung mehrerer Softwareeinheiten geschieht ausschließlich über die Schnittstelle dieser Einheiten. Für die Implementierung ist es ausreichend, die Schnittstellen aller benutzten Einheiten zu kennen. Das Ausprogrammieren verschiedener Softwareeinheiten kann völlig unabhängig voneinander geschehen, wenn die Schnittstellen festgelegt sind. Das bedeutet, daß die Softwareeinheiten parallel von mehreren Personen gleichzeitig entwickelt werden können. Das führt zu einer Effizienzsteigerung, wenn die Schnittstellen festgeschrieben werden und sich nicht oder kaum ändern.

Um die Vergleichbarkeit der Strukturierungsmechanismen der einzelnen untersuchten Sprachen zu gewährleisten, wird im nachfolgenden Abschnitt zunächst eine grafische Notation zur Beschreibung von Sichtbarkeitsbereichen eingeführt. Diese wird auch im Kapitel 5 eingesetzt, um die dort präsentierten Strukturierungsmechanismen darzustellen und mit den hier beschriebenen in Beziehung zu setzen.

3.1 Notation

Mit der in diesem Abschnitt eingeführten Notation und den darauf aufbauenden Untersuchungen wird ein Ziel verfolgt. Es soll aus den vorgestellten Strukturierungsmechanismen der hier behandelten Sprachen Ideen und Konzepte für die in TLMIN implementierten Strukturierungsmittel, die in Kapitel 5 behandelt werden, gezogen werden. Die Notation sichert dabei die Vergleichbarkeit der verwendeten Konzepte zu.

Zum Verständnis und zur Klarheit der nachfolgenden Diskussion werden an dieser Stelle zunächst einige Begriffe definiert, die im folgenden Benutzung finden (vgl. Abbildung 3.1).

Namen (auch Bezeichner genannt) dienen dazu, Objekte zu identifizieren und auf sie zuzugreifen. Nur benannte Objekte können auch referenziert werden. Die Möglichkeit, Namen für Objekte zu vergeben, ist die Grundvoraussetzung für die Diskussion über Sichtbarkeitsbereiche. Es ist für diese Untersuchung jedoch unerheblich, ob es für ein Objekt mehrere Namen gibt (Alias), welchen Typ das benannte Objekt hat bzw. ob es ein Typ ist.

Sowohl in der grafischen Notation, als auch in der Diskussion um Sichtbarkeitsbereiche wird nicht zwischen Deklaration und Definition eines Bezeichners unterschieden, wie dies z.B. in C getan wird. Das ist darin begründet, daß hier nur die Einführung von Namen betrachtet wird, die von beiden Operationen durchgeführt wird. Die darüber hinaus anhaftende Semantik ist für die Betrachtung der Systemstrukturierung nicht relevant.

Modulen nicht ausgeschlossen.

Sichtbarkeitsbereiche definieren die Menge an sichtbaren, d.h. über den Namen referenzierbaren Objekten innerhalb eines Ausdrucks oder einer Anweisung. Sichtbarkeitsbereiche werden gewöhnlich innerhalb von Blöcken definiert [Louden 1994].

Gültigkeitsbereiche von Bezeichnern erstrecken sich von ihrer Einführung über eventuell geschachtelte Blöcke bis zum Ende des Blocks, d.h. bis zur Grenze des Sichtbarkeitsbereichs[3]. Dies wird in Abbildung 3.2 anhand der in Abbildung 3.1 vorgestellten grafischen Notation schematisch verdeutlicht. Mit Gültigkeitsbereich ist abweichend zu [Louden 1994] gerade der Bereich gemeint, in dem der entsprechende Bezeichner erreichbar ist. Durch Namensgleichheit verdeckte Bezeichner befinden sich also nicht im Gültigkeitsbereich.

Lebensdauer bezeichnet den Zeitraum, in dem das Objekt oder der Wert, der über einen Bezeichner referenziert wird, vorhanden ist. Mit der Notation aus Abbildung 3.1 wird über die Lebensdauer keine Aussage getroffen. Das mit einem Namen assoziierte Objekt kann auch dann noch „leben“, wenn der Name nicht mehr gültig ist. Das ist z.B. bei Bezeichnern innerhalb eines Moduls bei Modula-2 der Fall. Lebensdauer ist ein Begriff, der sich auf tatsächlich existierende Objekte bezieht, für die auch Speicherplatz reserviert wird. Z.B. kann bei einer Verzweigungsmarke (*case label*) nicht von Lebensdauer gesprochen werden. Der wesentliche Unterschied zwischen Lebensdauer und Sichtbarkeit ist demnach: Die Lebensdauer betrifft die Eigenschaft und die Behandlung eines Objekts zur Laufzeit, ist also eine dynamische Größe. Die Sichtbarkeit beschreibt statisch die Erreichbarkeit von Namen, die auf Objekte verweisen.

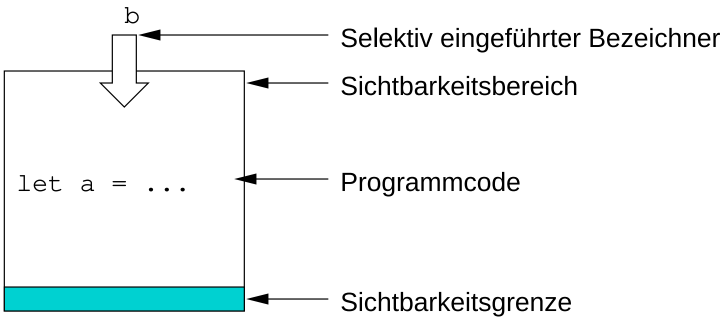


Abbildung 3.1: Grafische Notation zur Beschreibung von Sichtbarkeitsbereichen.

Über spezielle Sprachmechanismen gibt es in vielen Programmiersprachen die Möglichkeit, den Sichtbarkeitsbereich explizit einzuschränken bzw. selektiv Bezeichner in den Sichtbarkeitsbereich einzuführen. Die durch die Sichtbarkeitsbereiche definierte Menge an Bezeichnern kann demnach auf kontrollierte Weise sowohl zu- als auch abnehmen. Dem ist in der hier verwendeten grafischen Notation Rechnung getragen. Die Notation erlaubt es nicht, die Lebensdauer eines

[3] Ein Sichtbarkeitsbereich, der keine Sichtbarkeitsgrenze besitzt, kann alle Bezeichner seines übergeordneten Bereichs sehen und gibt alle hinzugefügten Bezeichner an diesen weiter. Er ist damit überflüssig.

Bezeichners darzustellen. Ihre beschreibende Wirkung bezieht sich ausschließlich auf statische Aspekte.

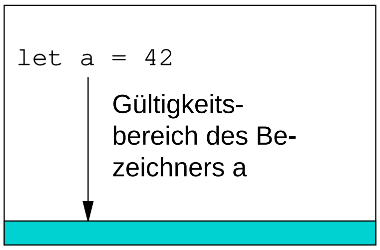


Abbildung 3.2: Gültigkeit eines Bezeichners.

Die hier präsentierte Notation ist bewußt einfach gehalten. Das Hinzufügen neuer Bezeichner wird z.B. durch Wiedergabe des entsprechenden Programmcodes verdeutlicht und nicht durch weitere grafische Elemente. Die Vergleichbarkeit verschiedener Sichtbarkeitssituationen ist damit nicht unbedingt gegeben, der Schwerpunkt der Notation liegt jedoch auf der Untersuchung von Sichtbarkeitsgrenzen und deren Lage sowie auf Bezeichnern, die diese Grenzen implizit oder explizit durch Sprachkonstrukte gesteuert durchdringen. Es wurde daher darauf verzichtet, in den nachfolgenden Abschnitten alle sichtbarkeitsrelevanten Konstrukte einer Sprache zu untersuchen und in der grafischen Notation abzubilden.

3.2 C++

C++ ist eine Sprache, die sich seit ihrer Einführung in einer andauernden Entwicklung befindet. Dies dokumentieren nicht zuletzt die verschiedenen Standardentwürfe der ANSI X3J16- und ISO WG21-Organisationen. Grundlage für diesen Abschnitt stellt die Referenz in [Stroustrup 1991] dar.

Wie in [Stroustrup 1991] beschrieben, nimmt C++ für sich in Anspruch, die von C bekannten Defizite bei der Programmierung im Großen zu beseitigen, in dem es Modularisierung und streng typisierte Schnittstellen anbietet. Auch die Programmierung im Team wird durch diese Konstrukte unterstützt.

In C++ stehen auf verschiedenen Abstraktionsebenen unterschiedliche Konstrukte zur Verfügung, die im folgenden kurz besprochen und im Hinblick auf die damit verbundenen Manipulationen der Sichtbarkeitsbereiche untersucht werden. Es stehen vier grundlegende Sichtbarkeitsbereichskonstruktionen (Blöcke, Funktionen, Klassen und Dateien) zur Verfügung. Ergänzt werden sie durch das im neuesten Standardentwurf vorgestellte Konstrukt zur expliziten Manipulation von Namensräumen (*namespace*).

3.2.1 Blöcke

In C++ bilden Blöcke neben Klassen eine grundlegende Strukturierungseinheit. Sie haben die für blockstrukturierte Sprachen übliche in Abbildung 3.3 gezeigte Semantik, daß alle außerhalb definierten und deklarierten Bezeichner innerhalb des Blocks sichtbar sind, alle innerhalb definierten jedoch außerhalb nicht.

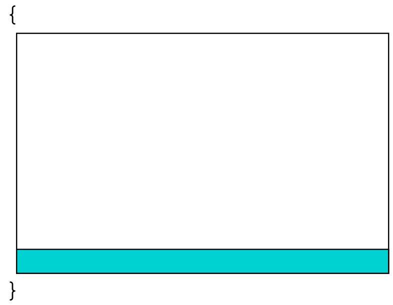


Abbildung 3.3: Sichtbarkeitsbereich eines Blocks in C++.

3.2.2 Funktionen

Funktionen sind ein Mechanismus, um C++-Programme in kleinere Teile zu zerlegen und damit in Verbindung mit Parametrisierung die Wiederverwendbarkeit zu fördern. Im Gegensatz zu anderen objektorientierten Programmiersprachen können in C++ Funktionen auftreten, die nicht an Klassen gebunden sind.

Funktionen erlauben die Übergabe einer beliebigen Anzahl von Parametern, die auf verschiedene Weise erfolgen kann:

Per Wert: Die Funktion erhält eine Kopie des übergebenen Wertes. Änderungen des Wertes haben keine Auswirkungen außerhalb der Funktion.

Per Zeiger: In diesem Fall wird keine Kopie angelegt, sondern direkt auf der betroffenen Variable gearbeitet.

Per Referenz: In der Wirkung ist die Übergabe per Referenz mit der per Zeiger identisch. Im Unterschied zu dieser wird die Variable jedoch syntaktisch wie ein Wertparameter behandelt, es wird also ohne Inhalts- und Adreßoperator gearbeitet.

Für Parameter können Standardwerte vorgegeben werden. Funktionen lassen sich somit mit variabler Parameteranzahl aufrufen. Dies erlaubt es, eine Funktion um Parameter zu erweitern, ohne daß alle Funktionsaufrufe angepaßt werden müssen. Die fehlenden Argumente werden mit den angegebenen Standardwerten gefüllt.

Funktionen können überladen werden. Das bedeutet, daß es Funktionen mit gleichem Namen, jedoch mit Parametern unterschiedlichen Typs und in unterschiedlicher Anzahl geben kann. Anhand der Funktionssignaturen entscheidet der Compiler, welche Funktion tatsächlich aufgerufen wird.

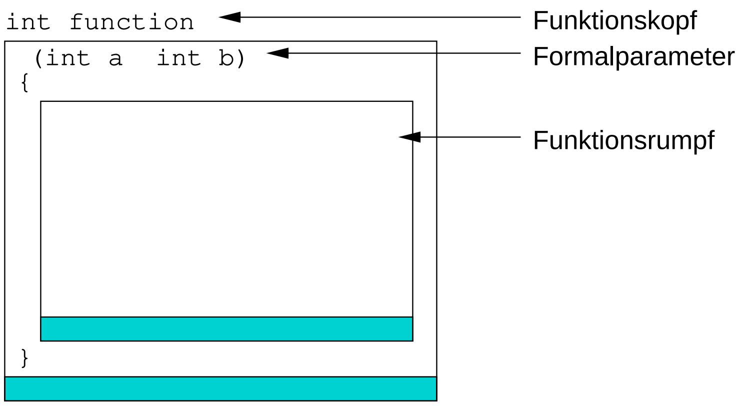


Abbildung 3.4: Sichtbarkeitsbereich einer Funktion in C++.

Bezogen auf die in Abbildung 3.4 dargestellte Sichtbarkeitssituation einer Funktion gibt es eine Ausnahme. Dabei handelt es sich um Marken (*labels*), die an beliebiger Stelle innerhalb der Funktion definiert werden können, jedoch im gesamten Funktionsrumpf sichtbar sind. Ihr Gültigkeitsbereich erstreckt sich auf einen Bereich vor ihrer Definition.

Aus Abbildung 3.4 geht hervor, daß der Funktionskopf außerhalb des äußersten abgeschlossenen Sichtbarkeitsbereichs liegt. Der Funktionsname ist deshalb für nachfolgende Ausdrücke auf dieser Sichtbarkeitsebene erreichbar. Die Parameter sind nur innerhalb der Funktion sichtbar, sie werden durch den sie umschließenden Sichtbarkeitsbereich nicht nach außen weitergegeben. Der Funktionsrumpf entspricht einem Block, wie er in Abschnitt 3.2.1 vorgestellt wurde. Die gleiche Sichtbarkeitssituation könnte auch mit nur einem Block erreicht werden, an dessem Anfang die Parameternamen bekannt gemacht werden. Konzeptuell erfolgt die Bindung der Aktual- an die Formalparameter jedoch vor Betreten des Funktionsrumpfes [Kernighan und Richie 1990].

3.2.3 Klassen

Klassen sind in C++ benutzerdefinierte Typen. Sie stellen das auf der höchsten Abstraktionsebene angesiedelte Strukturierungskonzept der Sprache dar[4]. Abbildung 3.5 zeigt das Szenario der Sichtbarkeitsbereiche einer Klasse. Klassen können geschachtelt werden.

Durch die verwendete Notation kann nicht die volle Semantik dargestellt werden. Z.B. können die unter *private* aufgeführten Bezeichner nur in *friend* Klassen verwendet werden. Im Zusammenhang mit Vererbung existiert in C++ ein weiterer Zugriffsspezifizierer. So können Objekte mit *protected* nur dem Zugriff von abgeleiteten Klassen, nicht jedoch von benutzenden Klassen zugänglich gemacht werden. Die Schlüsselworte *public*, *private* und *protected* geben an, welche Klasse die spezifizierten Bezeichner vererben, importieren bzw. benutzen dürfen. Weiterhin kann die Definition von Mitgliedern einer Klasse textuell getrennt von der Deklaration erfolgen. Von diesem Tatbestand wurde hier abstrahiert. Syntaktisch kann die Definition einer Mitgliedsfunktion mit Hilfe des Resolutionsoperators *::* außerhalb der Klassendeklaration formuliert werden.

[4]Templates, die vom Abstraktionsniveau noch höher als Klassen angeordnet werden müßten, werden in dieser Arbeit nicht betrachtet.

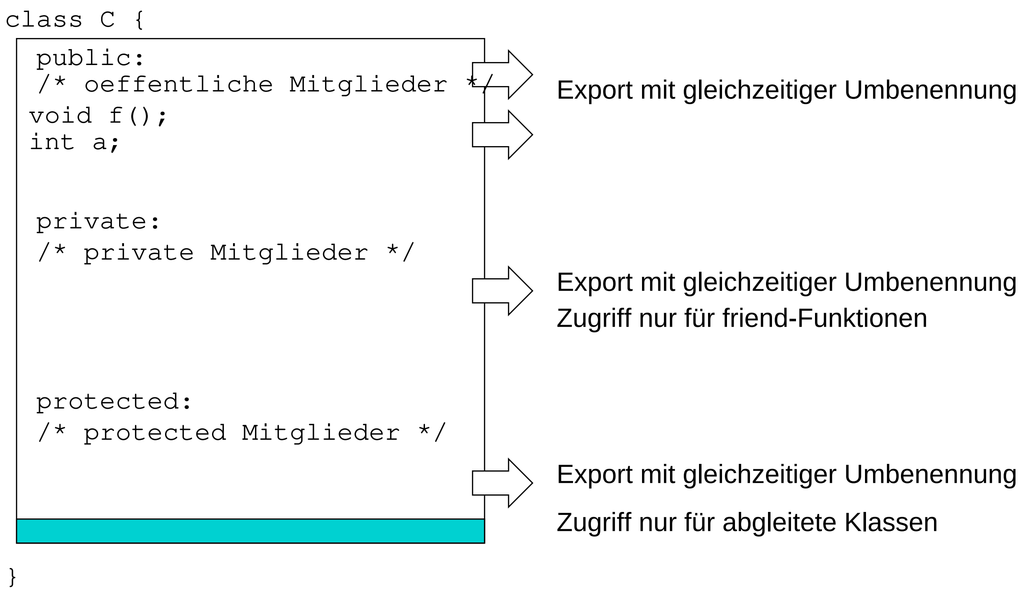


Abbildung 3.5: Sichtbarkeitsbereiche einer Klassendeklaration und -definition in C++.

Die Deklaration einer Klasse folgt damit streng dem Prinzip der Lokalität. Die Definition, also die Implementierung, kann jedoch verstreut über beliebig viele Dateien erfolgen.

Eine Klassendeklaration führt eine Reihe von neuen Namen ein. Außerhalb einer Klassendeklaration kann jeder Bezeichner der Klasse durch Qualifizierung erreicht werden, sofern der Zugriff erlaubt ist. Spezielle syntaktische Konstrukte zum Einführen von Namen existieren für Klassen nicht; benötigte Bezeichner werden bei Bedarf benutzt. Bei der Verwendung von Vererbung werden die in den Basisklassen enthaltenen Namen sichtbar. Abbildung 3.6 zeigt eine abgeleitete Klasse mit ihrer Basisklasse. Klasse *abgeleitet* hat die zusätzlichen Mitglieder *a* und *b*. Beide Bezeichner wurden durch Angabe der Basisklasse in den Sichtbarkeitsbereich des Klassenrumpfes eingeführt.

```
class basis {
public:
  int a, b;
};

class abgeleitet : public basis {
public
  int c;
};
```

Abbildung 3.6: Abgeleitete Klasse in C++.

Abweichende Situationen entstehen, wenn Datenattribute oder Methoden der Klasse als statisch (*static*) deklariert sind. Statische Datenattribute sind pro Klasse nur einmal vorhanden. Somit existieren sie für alle Instanzen der Klassen nur einmal. Ihre Eigenschaft, statisch zu sein, hat keinen Einfluß auf ihre Sichtbarkeit. Anders bei statischen Methoden: Sie unterscheiden sich von nichtstatischen Methoden dadurch, daß es ebenfalls nur eine Methode für alle Instanzen der Klasse gibt, d.h., daß sie keine implizite Selbstreferenz auf die Instanz (*this pointer*) besitzen. Das bedeutet auch, daß sie sich nur auf ebenfalls statische Attribute und Methoden der Klasse beziehen können. In diesem Fall hat die Benutzung von *static* Einfluß auf die Sichtbarkeit der betroffenen Methode. Dies kann in der hier benutzten Notation nur unzureichend dargestellt werden.

3.2.4 Dateien

C++-Programme werden üblicherweise nicht in einer großen Datei abgelegt, sondern in mehrere kleinere Dateien, auch als Module bezeichnet, aufgeteilt. Die Aufteilung in Schnittstellen- und Implementationsdateien ist willkürlich und wird durch den Übersetzer nicht erzwungen. Demzufolge ist die Aufteilung in Schnittstellendateien (C++ Header-Dateien) und in Implementationsdateien reine Programmiererdisziplin.

Das Handhaben der Programmdateien wird durch Präprozessor-Anweisungen geregelt. Die angegebenen Dateien werden durch den Präprozessor textuell expandiert, so daß der Compiler stets nur einen kontinuierlichen Zeichenstrom verarbeitet. Dateien legen jedoch die Sichtbarkeit der darin definierten und deklarierten Bezeichner fest. So sind mit *static* definierte Bezeichner nur innerhalb der Datei sichtbar, in der sie definiert sind. Entsprechend können mit *extern* definierte Namen dateiübergreifend referenziert werden.

3.2.5 Namensräume

In dem neuesten Standardentwurf für C++ existiert mit dem *namespace* Konstrukt die Möglichkeit, Namen, die anderenfalls global wären, in Namensräumen zusammenzufassen. Zu diesen globalen Namen zählen z.B. alle Bezeichner, die auf der obersten Sichtbarkeitsebene einer Datei deklariert oder definiert werden. An anderer Stelle können diese Namen über eine *using* Klausel explizit eingeführt und benutzt werden, wobei dies qualifiziert oder unqualifiziert geschehen kann. Namensräume können geschachtelt werden.

Namensräume konstruieren neue Sichtbarkeitsbereiche. Sie führen neue Namen ein. So wird durch die folgende Anweisung ein neuer Sichtbarkeitsbereich aufgebaut. Die darin definierten Namen sind in ihrer Sichtbarkeit jedoch nicht eingeschränkt:

```
namespace A {
        class String { ... };
        void f(String);
}
```

Was in dieser Anweisung geschieht, ist, daß die Klasse *String* außerhalb der Namensraumdefinition durch

```
A::String
```

benutzt werden kann, somit einen weiteren kontextabhängigen Namen besitzt. Die Klasse hat innerhalb der Namensraumdefinition den Namen *String*, der nach seiner Deklaration referenziert werden kann. Alternativ kann mit

```
using A::String
```

die Klasse *String* unter ihrem unqualifizierten Namen in einem anderen Namensraum verfügbar gemacht werden.

Bezeichner können ähnlich wie bei Klassen auch außerhalb des Namensraums definiert werden, indem sie mit Hilfe des Resolutionsoperators formuliert werden. Sie werden vom Compiler so behandelt, als ob sie innerhalb des Blockes geschrieben wurden. Die Möglichkeit, Definitionen außerhalb des Namensraums zu formulieren, erschwert es, die Übersicht in einem Programm zu behalten, und verstößt gegen das Prinzip der Lokalität.

3.3 Modula-2

Modula-2 ist eine blockstrukturierte Programmiersprache, die von Pascal abstammt und u.a. ein Modulkonzept einführt. Dieses Modulkonzept bzw. die damit einhergehenden Auswirkungen auf die Sichtbarkeit von Bezeichnern werden in diesem Abschnitt genauer untersucht. Funktionen und Blöcke dienen in Modula-2 ebenfalls der Systemstrukturierung, werden hier jedoch aufgrund der Ähnlichkeit zu Funktionen und Blöcken in C++ nicht betrachtet.

3.3.1 Module

Module sind das wichtigste Abstraktionsmittel von Modula-2 [Cin et al. 1986]. Sie erlauben es, ein Programm in voneinander unabhängige und separat übersetzbare Einheiten zu zerlegen. Sie lassen sich in einen Definitions- und einen Implementationsteil aufspalten. Der Definitionsteil erlaubt es, Module zu programmieren, ohne daß die Implementation der referenzierten Module bereits existieren muß.

Abbildung 3.7 zeigt das Sichtbarkeitsszenario eines Moduls. Die Menge an Namen, die ein Modul exportiert, wird bei einem externen Modul durch das Definitionsmodul bestimmt. In einem internen Modul, wie es in Abbildung 3.7 dargestellt ist, geschieht dies durch die Angabe von Namen in der *export*-Klausel.

Module können geschachtelt werden. Für die Sichtbarkeit der Bezeichner ergibt sich dadurch jedoch keine neue Situation. In der grafischen Notation kann diese Schachtelung auf natürliche Weise durch Schachtelung der entsprechenden Objekte dargestellt werden (s. Abbildung 3.8). Daraus ergibt sich, daß ein lokales Modul nur die Bezeichner importieren kann, die im umschliessenden Modul sichtbar sind. Die Module *C* und *D* in Abbildung 3.8 können keine Bezeichner direkt aus *A* oder *B* importieren, sondern nur, wenn diese Bezeichner auch von Modul *E* importiert werden. Geht man von einem Modell aus, in dem sich alle verfügbaren, nicht geschachtelten Module in einem Namensraum befinden, so können Module nur aus dem direkt sie umgebenden

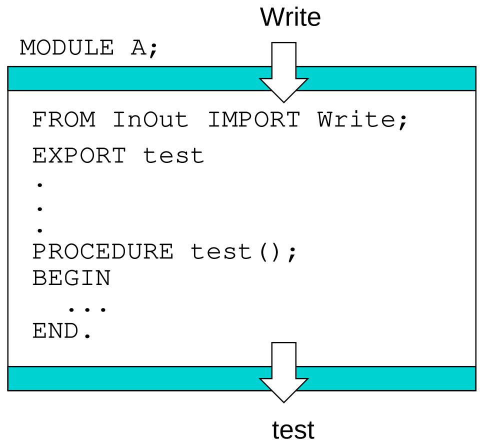


Abbildung 3.7: Sichtbarkeitsbereiche eines Moduls in Modula-2.

Namensraum importieren. Auf dem globalen Namensraum ist keine Reihenfolge der Namen definiert, was zyklische Abhängigkeiten prinzipiell möglich macht. In Darstellung 3.8 sind die Module *A* und *B* zyklisch voneinander abhängig.

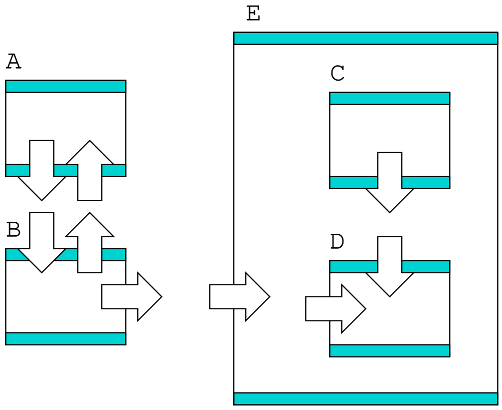


Abbildung 3.8: Geschachtelte Module mit Importbeziehungen in Modula-2.

Modula-2 bietet auf Sprachebene ein strukturiertes Konzept, das Modulkonzept, an, das die Sichtbarkeit von Namen eindeutig regelt. Für die Programmierung großer Systeme fehlt jedoch ein weiteres Konzept, um Module zu verwalten und zu organisieren. Geschachtelte Module können diese Aufgabe nur unzureichend erfüllen, da das zugrundeliegende Benennungsschema unflexibel ist. Auch Umstrukturierungen lassen sich schon durch die auch rein textuelle Schachtelung von Modulen schwer bewerkstelligen.

3.4 Napier88

Bei Napier88 handelt es sich wie bei TLMIN um eine persistente Programmiersprache mit einem polymorphen Typsystem und der Unterstützung von Funktionen höherer Ordnung [Morrison et al. 1988]. Im Kontext von Napier88 wurde an der Universität von St. Andrews ein Schwerpunkt auf die Untersuchung der Implikationen und Möglichkeiten einer persistenten Sprache auf Softwareentwicklungsumgebungen gelegt [Morrison et al. 1994; Morrison et al. 1993]. Die genaue Examinierung der Sprachkonstrukte, die Sichtbarkeitsbereiche berühren, wie z.B. Blöcke, Funktionen usw., kann hier aufgrund der Ähnlichkeit zu TLMIN entfallen.

Das Augenmerk dieses Abschnittes wird auf die in [Dearle 1989] in Napier88 eingeführten Umgebungen (*environments*) gelegt. Zwar weisen sie ebenfalls Ähnlichkeiten mit den in Abschnitt 5.3 vorgestellten Umgebungen des TLMIN Systems auf, unterscheiden sich von diesen jedoch in einigen wichtigen Punkten.

3.4.1 Umgebungen

Umgebungen sind ein fester Bestandteil der Programmiersprache Napier88 und Objekte erster Klasse. Umgebungen ermöglichen das Speichern und Verwalten von Bindungen durch die folgenden Operationen:

- Leere Umgebung erzeugen.
- Umgebung um eine Bindung erweitern.
- Den Wert einer Bindung extrahieren.
- Eine Bindung aus einer Umgebung entfernen.

Umgebungen können in Ausdrücken dazu benutzt werden, Namen einzuführen und zu benutzen, wie das folgende Beispiel zeigt:

```
use new as a:int in
begin
  writeint(a)
end
```

Der Name *a* wird in der nachfolgenden Sequenz benutzt. Er wurde aus der Umgebung *new* projiziert. Durch die Angabe der Signatur bleibt dieser Mechanismus typsicher. Die Typkorrektheit kann jedoch nicht statisch zugesichert werden. Bei Auftreten einer *use*-Klausel muß die unter dem angegebenen Namen aufgefundene Bindung gegen die Signatur geprüft werden. Dies impliziert, daß für jede Bindung einer Umgebung dynamische Typinformation bereitsteht. Beim Fehlschlagen des Typtests muß eine Ausnahme ausgelöst werden.

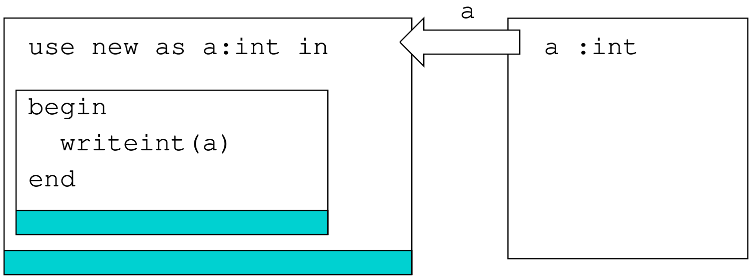


Abbildung 3.9: Sichtbarkeitssituation bei der Einführung eines Bezeichners aus einer Umgebung.

Kapitel 4

Entwicklungsumgebung TLMIN

Ausgehend von einem idealen Modell eines Übersetzers werden in diesem Kapitel der Aufbau und die Architektur des TLMIN (*Tycoon Language minimal*) Systems und die Werkzeuge zur Softwareentwicklung bzw. die vom TLMIN System für deren Realisierung bereitgestellten Basisdienste vorgestellt. Dabei wird auf die besonderen Implikationen dieses Systems bei der Realisierung der in Kapitel 5 vorgestellten Strukturierungsmechanismen eingegangen.

Voraussetzung für die Realisierung der Modulverwaltung ist die in das TLMIN System integrierte reflektive Compilerschnittstelle, die es ermöglicht, wiederverwendbare Softwarekomponenten zu entwickeln, die auf der Anwendungsebene der Tycoon Architektur angesiedelt sind und auf das darunterliegende Tycoon System zugreifen. Die Compilerschnittstelle kann als ein Hilfsmittel zur Realisierung von Komponenten einer Entwicklungsumgebung bzw. von Werkzeugen, die direkt auf das System zugreifen müssen, wie z.B. die Modulverwaltung, dienen. Diese Werkzeuge können zur Laufzeit des TLMIN Systems angebunden werden, da sie nicht in den Compiler selbst integriert sind.

4.1 Begriffsverwendung

Die folgenden Begriffe bilden die Diskussionsgrundlage für dieses Kapitel.

Tycoon Architektur: Hierunter werden die für Tycoon Systeme charakteristische Schichtenbildung und die Verteilung verschiedener Komponenten auf diese Schichten verstanden (s. Abbildung 4.1). Die genaue Anordnung der einzelnen Komponenten auch im Hinblick auf die Einordnung in verschiedene Schichten kann zwischen unterschiedlichen Tycoon Systemen variieren. Die Tycoon Architektur legt weiterhin fest, welche Operationen und Repräsentationen an den Schnittstellen zwischen den Schichten Verwendung finden.

Tycoon System (Familie): Ein Tycoon System bezeichnet eine konkret existierende Implementierung der Tycoon Architektur. Diese konkret existierenden Implementierungen unterscheiden sich im wesentlichen durch die im Compiler Frontend (Sprachebene) realisierte Programmiersprache, können aber auch in anderen Aspekten wie der Objektspeicherimplementierung differieren.

Die Gesamtheit aller Implementierungen der Tycoon Architektur wird Tycoon System Familie genannt. Die Tycoon System Familie ist weiter in (Unter)familien aufgeteilt, die in bezug auf die Tycoon Architektur Gemeinsamkeiten aufweisen.

Der Begriff Tycoon System (im Singular) wird in dieser Arbeit dann gebraucht, wenn ein beliebiger Vertreter aus der Familie der Tycoon Systeme gemeint ist. Abweichend zu [Matthes 1993] ist mit Tycoon System also nicht ein bestimmtes System gemeint, welches TL (*Tycoon Language*) als Programmiersprache besitzt. Es wird versucht, eine klare Trennung zwischen Programmiersprache einerseits und implementierendem System andererseits vorzunehmen.

TL System: Dieser Begriff benennt ein Mitglied aus der Familie der Tycoon Systeme, genauer ein Mitglied der Unterfamilie der Systeme, die die Programmiersprache TL implementieren.

Alle Ausprägungen der Unterfamilie der TL Systeme besitzen die gleiche Sprachebene, können aber in anderen Schichten variieren. Dies verdeutlicht auch Abbildung 4.2.

TLMIN System: Weitere Mitglieder in der Familie der Tycoon Systeme sind die TLMIN Systeme. Analog zur Unterfamilie der TL Systeme existiert eine Unterfamilie von TLMIN Systemen. Jedes einzelne Element dieser Unterfamilie wird seinerseits TLMIN System genannt. Alle TLMIN Systeme haben ebenfalls die Eigenschaft, sich untereinander in der Sprachebene zu gleichen.

Ein TLMIN System realisiert die Programmiersprache TLMIN. In TLMIN Systemen werden ähnliche Konzepte wie in TL Systemen verwendet. Aus diesem Grund ist es möglich, daß die unter der Sprachebene liegenden Schichten eines TL und eines TLMIN Systems gleich sind. Dies gilt z.B. für die Implementation der Objektspeicherschicht. Zusätzlich zur Sprachebene umfaßt ein TLMIN System auch eine andere Ausprägung der Schicht der Arbeitsumgebung (TLMIN Arbeitsumgebung).

TL: Unter diesem Begriff versteht man die durch ein TL System angebotene Programmiersprache, die definiert ist durch eine Beschreibung der Syntax und der statischen und dynamischen Semantik in [Matthes 1993]. Synonym werden die Begriffe TL Sprache oder Sprache TL angewendet.

TLMIN: Analog der Definition für TL versteht man hierunter die durch ein TLMIN System angebotene Programmiersprache, deren abstrakte und konkrete Syntax sowie die statische Semantik in Form der Typregeln in [Schröder 1994] bzw. [Bremer 1996] angegeben sind. Die dynamische Semantik entspricht der Semantik der Sprache TL. Im Unterschied zu TL beschränkt sich TLMIN auf einen schmaleren Sprachkern.

Synonym zu TLMIN werden die Begriffe TLMIN Sprache oder Sprache TLMIN angewandt.

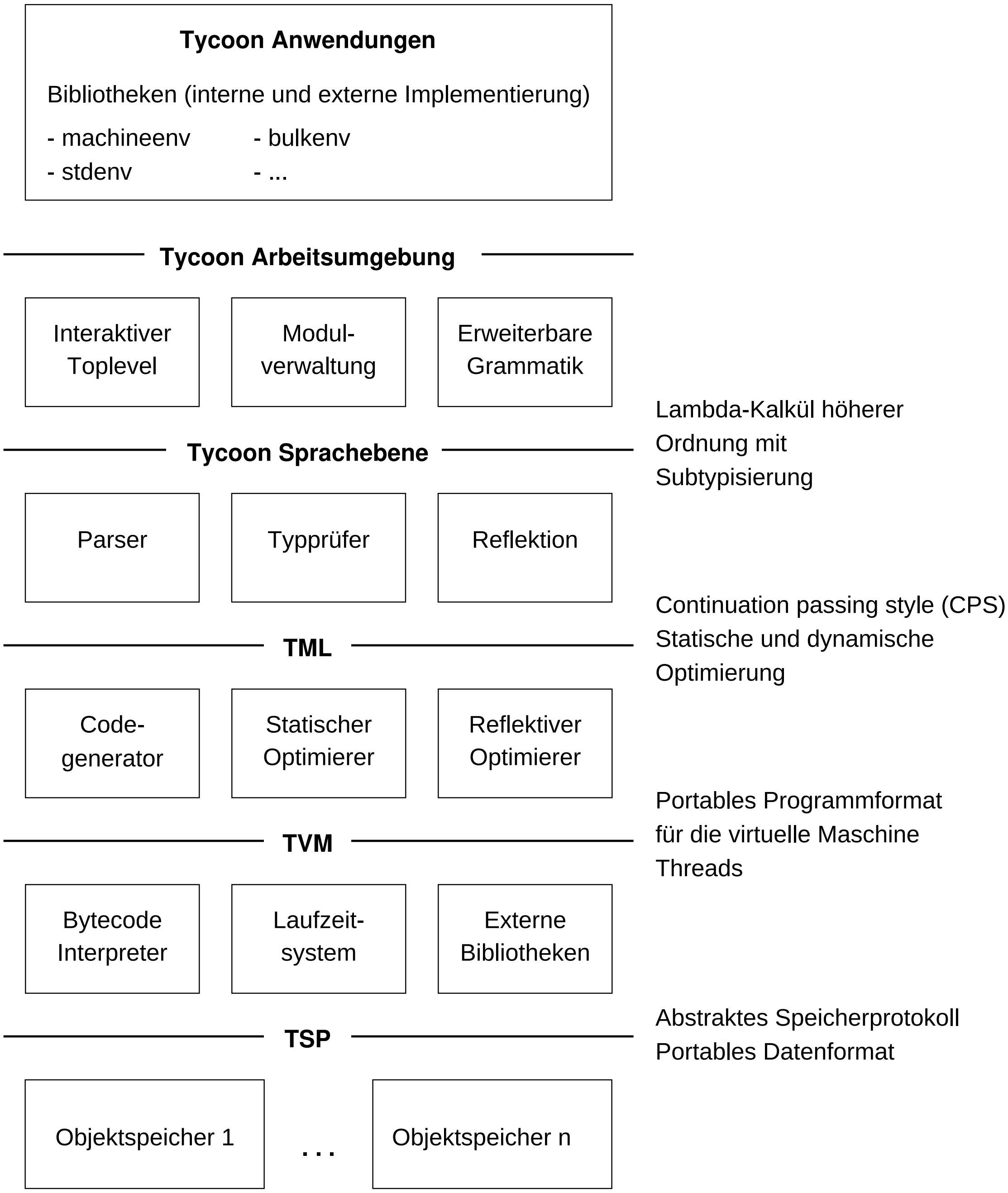


Abbildung 4.1: Überblick über die Tycoon Architektur.

Der Begriff Tycoon System steht also für eine Menge von Systemen, die wiederum zu unterschiedlichen Gruppen (Unterfamilien) zusammengefaßt werden können. In Abbildung 4.2 sind die Tycoon Systeme nach der implementierten Programmiersprache, der verwendeten Objektspeicherimplementierung und der ausführenden Tycoon Maschine partitioniert. Hervorgehoben ist die Menge der TLMIN Systeme, die eine mögliche Zusammenfassung einer Untermenge der Tycoon Systeme bildet.

Tycoon Systeme können nicht nur anhand ihrer unterschiedlichen Schichtenimplementierungen partitioniert werden. Auch andere Aufteilungen lassen sich finden, wie z.B. die Fähigkeit, nativen Maschinencode zu erzeugen. Im Rahmen dieser Arbeit wird diese Unterteilung herangezogen, um zwischen den Familien der TL und der TLMIN Systeme zu differenzieren und um zu betonen, daß die anderen Unterschiede für diese Arbeit nicht relevant sind.

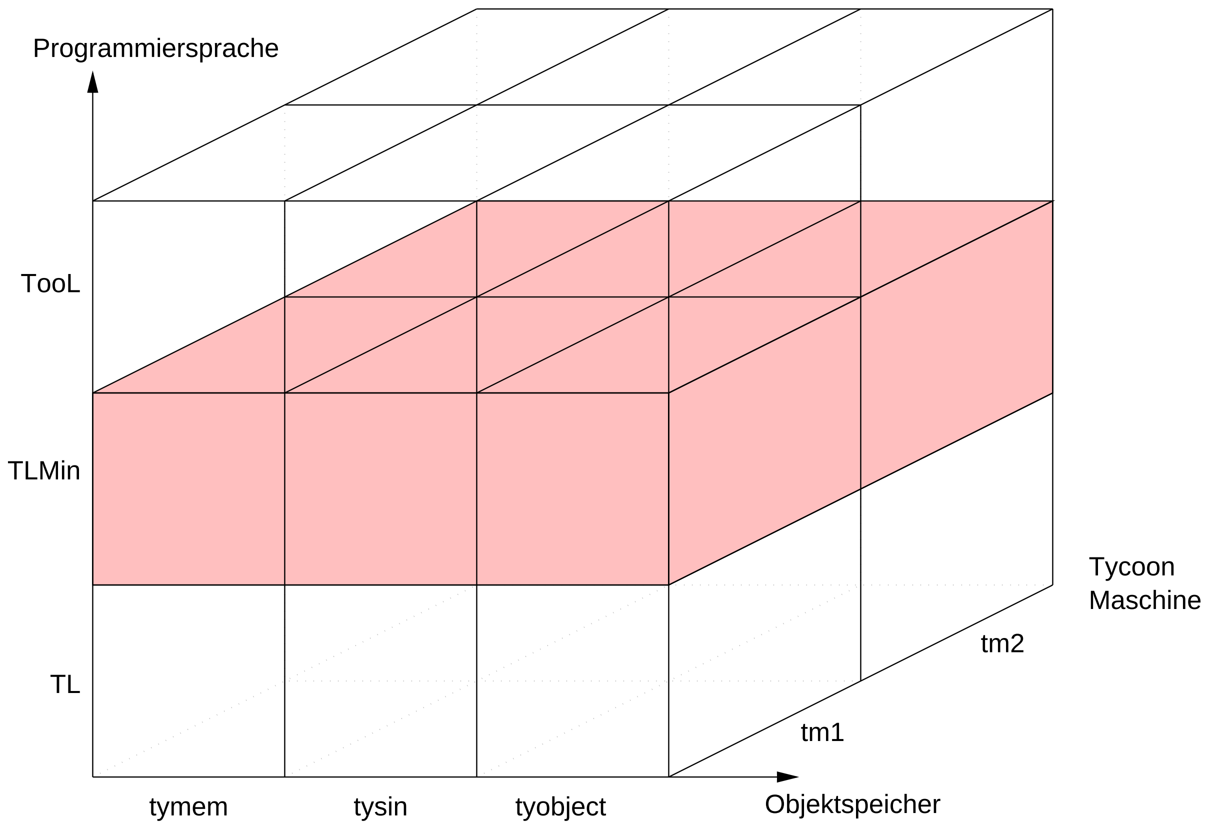


Abbildung 4.2: Mögliche Partitionierung der Tycoon Systeme.

4.2 TLMIN System

In diesem Abschnitt wird das in [Aho et al. 1988] beschriebene Phasenmodell eines Compilers präsentiert (vgl. [Schröder 1994]). Anhand dieses Modells werden Bezüge zum Compiler des TLMIN Systems hergestellt. Der Übersetzungsvorgang ist danach in Phasen unterteilt, die nacheinander ablaufen und auf den Repräsentationen der entsprechend vorgelagerten Phase aufbauen. Abbildung 4.3 verdeutlicht dies. Phasen und Repräsentationen des Modells finden eine Entsprechung in der in Abschnitt 4.3 behandelten Compilerschnittstelle bzw. in deren Funktionen und Datentypen.

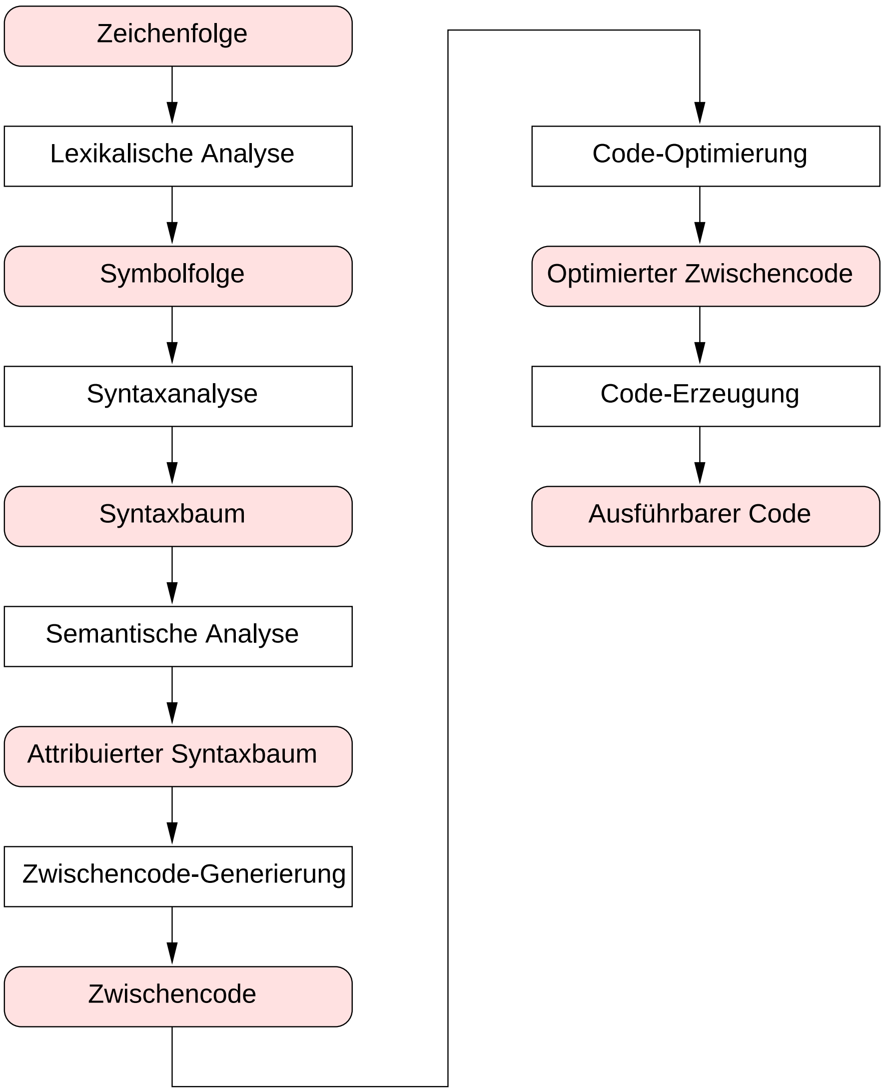


Abbildung 4.3: Idealisierte Darstellung der Compilerphasen und der Zwischenrepräsentationen.

4.2.1 Lexikalische Analyse

In der ersten Phase wird das als Zeichenstrom vorliegende Quellprogramm in eine Folge von Symbolen umgeformt (*scanning*). Zeichen bzw. Zeichenfolgen, die keine Bedeutung tragen, wie z.B. Tabulatorzeichen, Kommentare usw. werden bei der lexikalischen Analyse entfernt.

Die Vorschrift, nach der Zeichenfolgen zu Symbolen gruppiert werden, liegt üblicherweise in Form von regulären Ausdrücken vor. Mit Hilfe von Generatoren (*scanner generator*) ist die automatische Erstellung von Programmen zur lexikalischen Analyse nach vorgegebenen regulären Ausdrücken möglich.

Im TLMIN System wird ein ausprogrammierter Scanner verwendet, der mit einer festen Menge von Symbolen und deren Repräsentation arbeitet. Die Menge der Schlüsselwörter ist nicht festgelegt und kann sowohl erweitert als auch eingeschränkt werden [Schröder 1994].

4.2.2 Syntaxanalyse

Die Syntaxanalyse (*parsing*) prüft, ob die von der lexikalischen Analyse gelieferte Folge von Symbolen der durch eine Grammatik beschriebenen Programmiersprache genügt, d.h. ob mit der Grammatik die vorgegebene Folge von Symbolen erzeugt werden kann. Die Syntax einer Programmiersprache liegt oftmals als kontextfreie Grammatik vor, wobei nur eine bestimmte Menge der kontextfreien Grammatiken von existierenden Syntaxanalyseprogrammen effizient erkannt werden kann. Dabei handelt es sich um Grammatiken in LL- und LR-Form. Im TLMIN System wird eine LL(1)-Grammatik verwendet (LL-Form mit einem Zeichen Look-ahead) [Schröder 1994].

Auch für die Syntaxanalyse existieren Generatoren, die Programme zur Erkennung einer bestimmten Sprache automatisch erstellen (*parser generator*).

Im TLMIN System wird mit einer fest eingebauten Kerngrammatik gearbeitet, die jedoch über entsprechende Mechanismen erweitert werden kann. Ein inkrementeller Parsergenerator wird mit dieser Grammatik parametrisiert und erzeugt einen Parser, der die der Grammatik entsprechenden Ausdrücke akzeptiert.

4.2.3 Semantische Analyse

Die semantische Analyse verarbeitet die aus der Syntaxanalyse gewonnene Datenstruktur (*parse tree*) und reichert sie mit zusätzlicher Information an. Die folgenden vier Teilaufgaben sind Bestandteil der semantischen Analyse [Aho et al. 1988]:

Typüberprüfung: Der Compiler prüft die Typen der verwendeten Operatoren, Variablen und Funktionen und sichert nach erfolgreicher Prüfung die Kompatibilität bei Zuweisungen, Vergleichen und Parameterübergabe zu.

Überprüfung des Kontrollflusses: Für Operationen, die den Kontrollfluß des Programms verändern, muß sichergestellt werden, daß das Ziel der Kontrollflußänderungen existiert.

Überprüfung auf Eindeutigkeit: Hier wird die Eindeutigkeit der verwendeten Bezeichner, die Disjunktheit von Fallunterscheidungsmarken und Aufzählungstypelementen sichergestellt.

Auf Namen bezogene Überprüfungen: In einigen Programmiersprachenkonstrukten muß der gleiche Name mehrmals auftreten, z.B. Blocknamen in Ada. Diese Prüfung sichert die korrekte Verwendung dieser Namen zu.

Die wichtigste der aufgeführten Überprüfungen ist die Typüberprüfung. Bei den anderen Aufgaben handelt es sich überwiegend um routinemäßige Abprüfungen [Aho et al. 1988].

In TLMIN wird eine weitreichende semantische Analyse durchgeführt, die insbesondere die folgenden Zusicherungen für ein Programm gibt [Matthes 1993]:

- keine inkompatible Funktionsparameteranzahl oder inkompatiblen Funktionsparametertypen

- keine inkompatiblen Funktionsergebnistypen
- keine inkompatiblen Wertbindungen lokaler oder globaler Objektspeichervariablen
- kein Zugriff auf undefinierte oder uninitialisierte Variablen
- keine Zuweisungen an unveränderliche Objekte
- kein Zugriff auf nichtexistierende Tupel-, Rekord- oder Ausnahmekomponenten
- vollständige Fallmarken bei der vollständigen Fallunterscheidung
- keine *exit* Anweisung ohne passende *loop*
- keine *reraise* Anweisung ohne korrespondierendes *try*

4.2.4 Zwischencode-Generierung

Um den Compiler möglichst unabhängig von der erzeugten Zielsprache zu halten, erzeugt das Frontend (s. Abbildung 4.3) zunächst einen Zwischencode. Dieser wird anschließend vom Backend in Maschinensprache transformiert. Die Verwendung eines Zwischencodes hat im wesentlichen zwei Vorteile [Aho et al. 1988]:

- Durch Austausch des Backends kann der Compiler für eine andere Zielsprache verwendet werden. Im Tycoon System liegt ein weiterer Vorteil in der Austauschbarkeit des Frontends. In Abbildung 4.2 auf Seite 50 sind drei verschiedene Programmiersprachen (Frontends) dargestellt, die jeweils auf eine von zwei verfügbaren Tycoon Maschinen (Backends) aufsetzen können. Die Darstellung macht die freie Kombinierbarkeit verschiedener Frontends mit verschiedenen Backends deutlich. Z.B. läßt sich das TL Frontend mit dem *tm1*-Backend betreiben, ebenso kann es mit dem *tm2*-Backend zusammenarbeiten.
- Auf dem Zwischencode kann ein von der Ziel- und Quellsprache unabhängiger Optimierer arbeiten.

Die Einführung eines zusätzlichen Verarbeitungsschrittes und einer weiteren Repräsentation in die Compilerphasen hat neben architektonischen Gesichtspunkten hauptsächlich softwaretechnische Gründe. Die Komponenten des Front- bzw. Backends können jeweils mehrfach verwendet werden.

In TLMIN wird als Zwischencode Tycoon Maschinen Code eingesetzt, der in der Tycoon Machine Language (TML) formuliert ist. Es handelt sich hierbei um eine abstrakte persistente Zwischencoderepräsentation, die, wie im folgenden Abschnitt erläutert wird, sowohl für lokale als auch für globale Optimierungen eingesetzt werden kann. TML ist eng verwandt mit der Darstellung des Lambdakalküls im Fortsetzungsstil (*continuation passing style (CPS)*). TML kann als wertaufrufbasiertes Lambdakalkül mit Speichersemantik bezeichnet werden [Gawecki und Matthes 1996]. Der Hauptvorteil von TML gegenüber anderen Zwischencoderepräsentationen, wie z.B. dem Drei-Adreß-Code, liegt in der geringen Anzahl von Konstrukten, die zur Formulierung verwendet werden. Abbildung 4.4 zeigt die gesamte TML-Syntax und unterstreicht damit die Einfachheit und Kompaktheit dieser Darstellung.

$lit \in Lit$	Literale, Konstanten und Objektidentifikatoren
$t \in Temp$	Temporäre Variablen
$c \in Cont$	Fortsetzungsvariablen
$v \in Var$	Variablen (Bezeichner)
$prim \in Prim$	Primitive Operationen

v	$::=$	$t \mid c$	Variablen
val	$::=$	$lit \mid v \mid abs$	Werte
abs	$::=$	$\lambda(v_1 \dots v_n) app$	Abstraktion, $n \geq 0$
app	$::=$	$(val_0 val_1 \dots val_n) \mid (prim\ val_1 \dots val_n)$	Applikationen, $n \geq 0$

Abbildung 4.4: Abstrakte Syntax der Zwischensprache TML (aus [Gawecki und Matthes 1996]).

Wie die Sprache TLMIN unterstützt auch TML Funktionen höherer Ordnung und eignet sich daher besonders als Zwischenrepräsentation in einem Übersetzer.

4.2.5 Code-Optimierung

Übersetzer, die den Code in bezug auf sein Laufzeitverhalten oder seinen Speicherplatzbedarf verbessern, werden optimierend genannt. Die Code-Optimierung ist dabei lediglich eine Transformation des ursprünglichen Codes in eine äquivalente Darstellung.

Bei der Code-Optimierung lassen sich zwei grundsätzliche Arten unterscheiden. Das sind zum einen Optimierungen, die direkten Bezug auf die unterliegende Hardwareplattform nehmen und z.B. besondere vom Prozessor schnell ausgeführte Befehlsfolgen gegenüber anderen bevorzugen oder die Vergabe von Registern unter plattformspezifischen Gesichtspunkten vornehmen. In die zweite Kategorie fallen Optimierungen, die auf dem in Abschnitt 4.2.4 eingeführten Zwischencode arbeiten. Derartige Optimierungen können vom Optimierer des TLMIN Systems durchgeführt werden.

In [Kiradjiev 1994] wird ein Optimierer für TML vorgestellt. Das Grundprinzip besteht in der Anwendung der Reduktionsregeln des Lambdakalküls. Diese Regeln formalisieren die Mechanismen der Bindung von Bezeichnern, der Substitution von gebundenen Werten und der Applikation von Lambda-Termen [Pakendorf 1996].

4.2.6 Code-Erzeugung

Der letzte Schritt des Übersetzungsvorganges beinhaltet die Code-Erzeugung. Er transformiert den Zwischencode in eine äquivalente Darstellung der Zielsprache. Die Zielsprache ist bei den meisten Compilern plattformabhängig. Im Tycoon System besteht die Besonderheit, daß auch der Code der Zielsprache unabhängig von der zugrundeliegenden Plattform ist. Vielmehr übersetzt die Code-Erzeugung die TML-Repräsentation in die Darstellung als Bytecode, der von der virtuellen Maschine der Tycoon Architektur (Tycoon Virtual Machine) ausgeführt werden kann.

4.3 Compilerschnittstelle

Die Compilerschnittstelle des TLMIN Systems bietet Funktionen, Typen und Daten des (laufenden) Übersetzers bzw. darunterliegender Schichten an. Sie ermöglichen das Übersetzen von TLMIN Ausdrücken und den Zugriff auf interne Strukturen. Der Entwickler hat über die Compilerschnittstelle die Möglichkeit, direkt gegen den Compiler zu programmieren. Außerdem wird durch die Bereitstellung schon vom Compiler benutzter Module[1] verhindert, daß der Programmierer diese Module in einer anderen Version benutzt und damit die Korrektheit der Schnittstelle verletzt. Module müssen weiterhin nicht erneut in den Objektspeicher geladen werden.

Die hier vorgestellte Compilerschnittstelle dient nicht der direkten Benutzung durch den Anwendungsprogrammierer. Vielmehr bietet sie die Dienste des Übersetzers Programmen an, die im Anwendungsbereich angesiedelt sind und als Programmierwerkzeuge fungieren. Es lassen sich zahlreiche Werkzeuge vorstellen, die Gebrauch von einer Compilerschnittstelle machen können:

- Ein Debugger kann mit Hilfe der vom Übersetzer angebotenen Operationen und Daten Programme schrittweise oder bis zu einem bestimmten Punkt ausführen und aktuelle Variablenwerte ausgeben.

- Browser gehen über die Funktionalität eines Debuggers in bezug auf die verfügbaren Informationen hinaus, indem sie auch die vom Compiler generierten Zwischenrepräsentationen, wie z.B. den abstrakten Syntaxbaum, anzeigen können.

- Ein Profiler auf Sprachebene kann ähnlich einem Debugger die Ausführung eines Programms nach beliebigen Schritten anhalten und z.B. Informationen über die verbrauchte Prozessorzeit sammeln. Auch der Übersetzungsvorgang selbst kann auf diese Weise analysiert werden und z.B. die für jede Übersetzungsphase benötigte Zeit betrachtet werden.

- Mit Hilfe einer Modulverwaltung, wie sie in Abschnitt 4.4 eingeführt wird, lassen sich die programmiertechnischen Einheiten wie z.B. Module, Schnittstellen, Klassen usw. organisieren. Die Modulverwaltung sorgt insbesondere dafür, daß die erwähnten Einheiten getrennt voneinander und in bezug auf deren Abhängigkeiten in der korrekten Reihenfolge übersetzt werden und anschließend zur Ausführung gebracht werden können. Letzteres impliziert die Auflösung externer Referenzen.

- Eine grafisch orientierte Programmierwerkbank, wie sie in [Geisler 1995] vorgestellt wird, kann o.g. Werkzeuge integrieren und dem Programmierer eine komfortable Bedienoberfläche bieten.

[1]Hierunter fallen Module zur Behandlung von Massendatenstrukturen (list, hashtable usw.), Bildschirmausgaben (print) und Dateibehandlung (reader).

In dieser Arbeit wird mit der Realisierung der Compilerschnittstelle ein strikter add-on-Ansatz in der Anbindung von Programmierwerkzeugen verfolgt. Keines der in obiger Aufzählung erwähnten Werkzeuge wird, wenn es konform zur Compilerschnittstelle programmiert ist, in das TLMIN System integriert. Analog zur Diskussion in [Matthes und Schmidt 1991] ergeben sich daraus mehrere Vorteile:

- Auf die veränderten Anforderungen an Werkzeuge kann ohne Änderung des TLMIN Systems und damit schnell reagiert werden.
- Die Wartbarkeit solcher Werkzeuge wird aufgrund der losen Kopplung erhöht.
- Das TLMIN System behält seine schlanke Struktur und erhält sich dadurch seine Wartbarkeit.
- Es besteht keine Beschränkung in der Zahl, Güte und Verfügbarkeit von Werkzeugen. Neue Werkzeuge können jederzeit ohne Änderung des TLMIN System hinzugefügt werden.

4.3.1 Anforderungen

Die Aufgabe einer Schnittstelle ist es, Typen, Operationen und Daten bereitzustellen, die es dem Benutzer der Schnittstelle erlauben, die angebotene Funktionalität einfach und effizient zu verwenden. Dabei soll die Schnittstelle von implementierungsabhängigen Dingen abstrahieren. Dies ist um so schwieriger, je breiter das Spektrum der Benutzeranforderungen ist, da öffentliche Strukturen, die von einer Benutzergruppe verwendet werden, für eine andere uninteressant sind.

Mit der Compilerschnittstelle wird das Ziel verfolgt, dem Programmierer eine weitreichendere Kontrolle über das System zu geben. Insbesondere hat er durch die über die Schnittstelle angebotenen Umgebungen die Möglichkeit, den Übersetzungsvorgang zu parametrisieren. Ausdrücke lassen sich durch die Verwendung mehrerer Umgebungen in unterschiedlichen Kontexten übersetzen. Auch die Verwendung verschiedener Compilereinstellungen, wie z.B. Optimierer ein-/ausgeschaltet, beeinflußt die Übersetzung.

Dem Programmierer soll über die Compilerschnittstelle weiterhin die Möglichkeit gegeben werden, die einzelnen Übersetzerphasen klar voneinander zu trennen und die Ausführung nach jedem Schritt anhalten zu können. Die nach jedem Schritt erzeugten Zwischenrepräsentationen können persistent abgelegt werden und z.B. durch Werkzeuge analysiert werden.

Aus diesem Szenario ergeben sich zusammenfassend folgende Anforderungen:

- unterschiedlich abstrakte, aber dennoch kompatible Schnittstellen
- Bereitstellen von Zwischenrepräsentationen
- parametrisierbarer Übersetzungsvorgang
- Offenlegung der Übersetzerphasen

Um der Anforderung nach unterschiedlichen Abstraktionsgraden gerecht zu werden, wurden zwei Module entworfen. Beide Varianten unterstützen alle übrigen Anforderungen.

4.3.2 Vereinfachte Compilerschnittstelle

Die vereinfachte Schnittstelle arbeitet ausschließlich mit abstrakten Datentypen für die Zwischenrepräsentationen und die Hilfsstrukturen. Es kann deshalb nur über entsprechende Funktionen darauf zugegriffen werden. Die Übersetzung und Ausführung eines Ausdruckes läßt sich folgendermaßen beschreiben:

```
execute(context
  generateCode(context
    translate(context
      typecheck(context
        parse(context sourceFromString("let a = 42"))))))
```

Da der Compiler weitere Parameter als die Zwischenrepräsentationen benötigt, erhält man nicht die idealisierte Darstellung wie in [Bremer 1996] oder [Geisler 1995]. Um die Verwendung der Schnittstelle jedoch möglichst einfach zu halten, sind alle übrigen Parameter in einem weiteren Datentyp nach dem Vorbild des Grafikkontextes (*graphic context*) der *Xlib* Bibliothek zusammengefaßt [Nye 1993a; Nye 1993b].

Der Kontext umfaßt die Größen, mit denen der Übersetzer parametrisierbar ist:

- Schalter, die den Übersetzungsvorgang steuern, wie z.B. ein Boole'scher Wert, der angibt, ob beim Typüberprüfen ein Cache für bereits geprüfte Ausdrücke angelegt werden soll.
- Quelltext
- Abstraktion des Fehlerkanals, der Meldungen des Compilers aufnimmt.
- Grammatik
- Parser
- Initiale Umgebung
- Umgebung, die die Zwischenrepräsentationen der bereits übersetzten Ausdrücke enthält

In Anhang A ist die gesamte Schnittstelle aufgeführt.

4.3.3 Detaillierte Compilerschnittstelle

Eine zweite Schnittstelle, die parallel zur oben beschriebenen aktiv sein kann, bietet eine detaillierte Sicht auf den Compiler. Der erweiterte Funktionsumfang gestattet einen weitreichenderen Zugriff auf den Compiler und dessen Datenstrukturen. Anhang B gibt die Schnittstelle wieder.

Ähnlich der einfachen Compilerschnittstelle müssen vor Benutzung der Schnittstelle einige elementare Datenstrukturen, wie z.B. der Fehlerkanal, die Grammatik oder der Parser initialisiert bzw. erzeugt werden. Die folgenden Operationen erledigen dies:

```
let switches = tlCompiler.newSwitches()
let log = tlCompiler.newLog()
let var gram = tlCompiler.newGrammar()
let var parser = tlCompiler.newParser(gram)
let initEnv = tlCompiler.newInitialEnvironment()
let env = environment.new()
```

Einige Komponenten werden als veränderliche Bindung angelegt. Bei der Grammatik *gram* ist dies erforderlich, da diese zur Laufzeit erweitert werden kann. Der Parser *parser* ist zum Zeitpunkt dieser Anweisung noch nicht mit der konkreten Grammatik parametrisiert worden. Dies geschieht in einem weiteren, hier nicht gezeigten, Schritt. Die Umgebung *env* enthält die Repräsentationen bereits übersetzter Bezeichner (vgl. Abschnitt 5.3).

Ein typischer Übersetzungsvorgang[2], bei dem alle Zwischenrepräsentationen persistent gespeichert werden, kann wie folgt ausgeführt werden. An dieser Stelle wird ein Vorgriff auf die Abschnitt 5.3 vorgestellten Umgebungen gemacht. Es werden die Funktionen beschrieben, die zur Benutzung der Compilerschnittstelle notwendig sind. Die auf den Aspekt der Strukturierung fokussierten Eigenschaften werden in Abschnitt 5.3 behandelt.

```
let s = tlCompiler.sourceFromString("let a=42;")
let b = tlCompiler.parse(parser gram s log)
let t = tlCompiler.typeCheck(switches initEnv
                             env b log)
let env1 = environment.insertSyntaxTree(env t)
let c = tlCompiler.translate(switches initEnv
                             env1 t log)
let env2 = environment.insertTMLCode(env1 c)
let g = tlCompiler.generateCode(switches env2 c log)
let r = tlCompiler.execute(switches env2 g log)
let env3 = environment.insertRuntimeValue(env2 r)
tlCompiler.printResult(switches r log)
tlCompiler.printLogClear(log)
```

Die Funktionen der detaillierten Compilerschnittstelle sind so angelegt, daß sie nur mit den tatsächlich benötigten Parametern versorgt werden müssen. In obigen Beispiel wird zunächst für den als String vorliegenden Ausdruck ein vom Compiler zu verarbeitender Zeichenstrom *s* erzeugt. Dieser wird anschließend anhand der Grammatik vom Parser in den Symbolstrom *b* umgewandelt. Die semantische Analyse (*tlCompiler.typeCheck*) liefert den attribuierten Syntaxbaum *t*, der als Parameter für die Zwischencode-Generierung dient. Die Umgebung *env* wird um den attribuierten Syntaxbaum mit der Funktion *environment.insertSyntaxTree* erweitert und als Umgebung *env2* zurückgegeben. Sie steht damit weiteren Ausdrücken bei der semantischen

[2]Für diese Anweisungen wird angenommen, daß der Parser mit einer Grammatik parametrisiert wurde, die die Ausdrücke der Sprache TLMIN erkennt.

Analyse zur Verfügung. Die neue Umgebung *env2*, die im Gegensatz zu *env1* zusätzlich Typinformation für den Bezeichner *a* enthält, wird zusammen mit dem attribuierten Syntaxbaum von der nachfolgenden Funktion verwendet. Auch die ursprünglich verwendete Umgebung *env* steht weiterhin zur Verfügung, da keine der beteiligten Funktionen Seiteneffekte auf Umgebungen ausführen.

Die Zwischencode-Generierung (*tlCompiler.translate*) fügt Zugriffsinformation zu den in der Umgebung *env2* abgelegten Objekten des attribuierten Syntaxbaums für den zu übersetzenden Ausdruck hinzu und gibt den erzeugten TML-Code *c* zurück. Dieser dient der Erweiterung der Umgebung *env1* zur Umgebung *env2* mit der Funktion *environment.insertTMLCode)*. Auch hier steht die ursprüngliche Umgebung *env* weiterhin zur Verfügung.

Die letzten beiden Schritte bestehen in der Generierung des TVM Codes *g* mittels der Funktion *tlCompiler.generateCode* und in der Ausführung dieses TVM Codes durch die virtuelle Maschine mit der Funktion *tlCompiler.execute*. Der evaluierte Wert *r* wird ebenfalls zur Erweiterung der Umgebung *env2* zur Umgebung *env3* benutzt.

Tritt in einer der Phasen ein Übersetzungsfehler auf, so wird ein Ausnahmepaket ausgelöst. Die Meldungen des Compilers werden nicht direkt auf der Konsole ausgegeben, sondern in einen Fehlerkanal geschrieben. Dies erleichtert die Programmierung von Entwicklungsumgebungen, die die Fehlerinformation z.B. in einem Fenster ausgeben.

Einige Hilfsfunktionen unterstützen den Programmierer in der Untersuchung der erzeugten Zwischenwerte. So kann z.B. mit *tlCompiler.printEnvironment* eine Liste aller aktuell in der Umgebung befindlichen Bezeichner ausgegeben werden. Weiterhin können die Regeln der Grammatik, der Syntaxbaum, der TML Code und in Zusammenhang mit den Schnittstellen zu den Umgebungen die Signaturen einzelner Bindungen angezeigt werden.

4.4 Modulverwaltung

In den verschiedenen konkreten Ausprägungen des Tycoon Systems lassen sich unterschiedliche Ansätze zur Verwaltung von Softwareeinheiten wie z.B. Module, Klassen oder Bibliotheken identifizieren. In den in dieser Arbeit vorgestellten Systemen der TLMIN Familie wird keine Festlegung auf eine bestimmte Verwaltungskomponente getroffen. Vielmehr ist es dem Programmierer überlassen, eine seinen Wünschen entsprechende Implementierung auszuwählen. Dieser generische Ansatz wird weiter unterstützt durch die in Kapitel 5 vorgestellte flexible Methode, neue Strukturierungsmechanismen aus vorhandenen Basisdiensten aufzubauen.

Im Gegensatz zur allgemeinen Tycoon Architektur, wie sie in Abbildung 4.1 auf Seite 49 dargestellt ist, unterscheidet sich die konkrete Architektur des TLMIN Systems in den obersten Schichten. In Abbildung 4.5 sind die betreffenden Änderungen eingearbeitet.

Die Modulverwaltung befindet sich auf der Anwendungsebene und interagiert mit dem Compiler über die bereitgestellten Komponenten Compilerschnittstelle, Umgebungen und erweiterbare Grammatik. Durch die Realisierung der Modulverwaltung, stellvertretend für andere Werkzeuge, als Anwendungsprogramm wird erreicht, daß sowohl die Entwicklung und Implementierung als auch die Wahl und der Einsatz eines Werkzeuges ohne Änderung des Compilers vorgenommen werden kann.

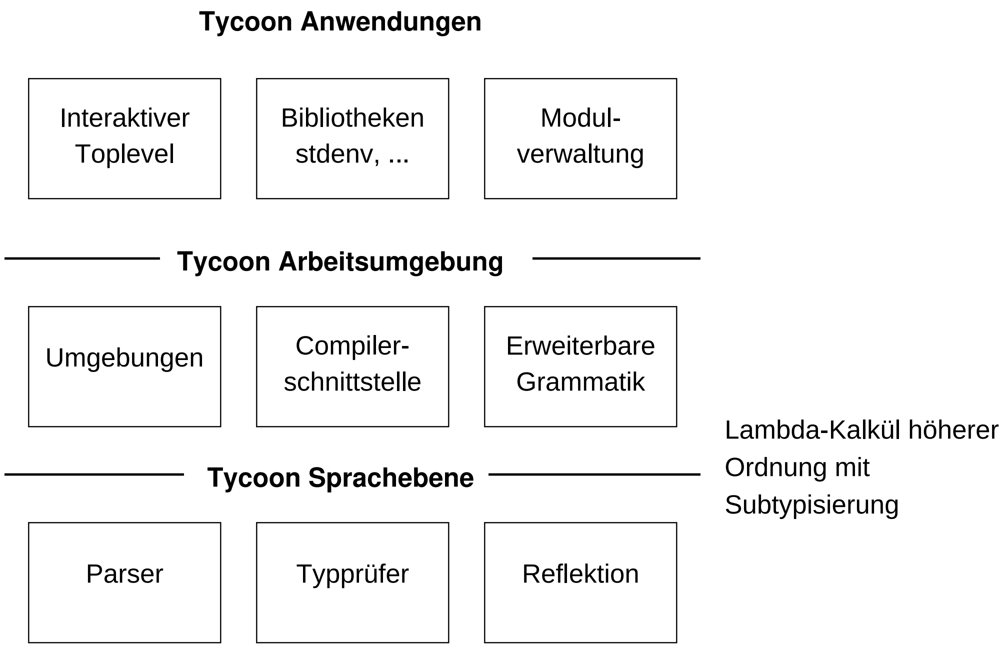


Abbildung 4.5: Überblick über die spezielle Architektur der TLMIN Systeme.

Die Aufgaben der Modulverwaltung bestehen darin, die Abhängigkeiten von Softwareeinheiten zu ermitteln und deren Systemgenerierung vorzunehmen. Mit Softwareeinheiten können hier z.B. Module und Schnittstellen gemeint sein. Durch die Kombination der in Kapitel 5 vorgestellten Basisprimitive zur Systemstrukturierung können auch andere Konstrukte aufgebaut und von der Modulverwaltung verwaltet werden. Dazu muß lediglich sichergestellt werden, daß die Modulverwaltung über die folgenden Punkte Kenntnis hat:

- Die Art und Weise, mit der die abhängigen Komponenten ermittelt werden.
- Die Methode, mit der die Bindung der Softwareeinheiten vorgenommen wird.
- Die Reihenfolge, in der die Bindung der Softwareeinheiten erfolgt.

Aus dem letzten Punkt der Aufzählung ergibt sich bereits eine weitere Aufgabe. Die Modulverwaltung muß nicht nur den Übersetzungsvorgang der Softwareeinheiten aufgrund der Abhängigkeiten steuern, sondern auch die Reihenfolge der Bindung bestimmen. Diese Reihenfolge richtet sich meist ebenfalls nach den Abhängigkeiten der Softwareeinheiten.

Im Kern stützt sich die Modulverwaltung auf die Verwaltung von Abhängigkeiten. In der zu dieser Arbeit angefertigten Implementierung wird dazu das in [Koch 1996b] entwickelte generische Werkzeug zur Modellierung von Abhängigkeitsgraphen verwendet. Es werden die in Abschnitt 5.2.1 und 5.2.2 vorgestellten Module und Schnittstellen unterstützt. Die Bindung der Komponenten wird auf Namensbasis durchgeführt. Damit werden nicht alle in Abschnitt 5.2.3 genannten Möglichkeiten ausgeschöpft. Die Implementierung selbst ist so flexibel gehalten, daß sie leicht an andere Verhältnisse, z.B. andere Strukturierungskonstrukte, angepaßt werden kann.

Kapitel 5

Systemstrukturierung in TLMIN

Die Untersuchung von Programmiersprachen und deren Mittel zur Systemstrukturierung in Kapitel 3 bildet den Ausgangspunkt für die Diskussion der Systemstrukturierungsmechanismen der Sprache TLMIN in diesem Kapitel. Die Aspekte der verschiedenen Maßnahmen zur Systemstrukturierung sind bereits in Kapitel 3 ausführlich beschrieben. In TLMIN wird der Ansatz verfolgt, dem Programmierer nicht ein festgelegtes Konzept zur Systemstrukturierung, wie z.B. Module oder Klassen, anzubieten. Stattdessen werden in TLMIN aufbauend auf einem geringen Satz an Basisprimitiven höhere, semantisch reichhaltigere Konzepte zur Systemstrukturierung realisiert. Dies gibt dem Programmierer eine größere Flexibilität und Freiheit in der Wahl der Strukturierungsmittel. Die adäquate sprachliche Anbindung kann über eine geeignete Syntaxerweiterung erfolgen und mit Werkzeugen unterstützt werden.

In diesem Kapitel werden zunächst die grundlegenden Mechanismen vorgestellt, die direkt in die Sprache TLMIN eingebaut sind. Sie bilden die Grundlage für höhere Strukturierungskonzepte, die in Abschnitt 5.2 beschrieben werden. Eine Modulverwaltung, wie sie in Abschnitt 4.4 vorgestellt ist, kann die daraus gebildeten Einheiten verwalten.

Abschnitt 5.3 stellt Umgebungen sowohl als weiteres Strukturierungsmittel als auch als Hilfsstruktur des Compilers vor. Der anschließende Abschnitt führt die vorgestellten Konzepte in einem Anwendungsszenario zusammen.

5.1 Manipulation von Sichtbarkeitsbereichen

Die wichtige Rolle, die Sichtbarkeitsbereiche und deren Manipulation bei der Programmierung im Großen spielen, wurde bereits in Kapitel 3 dargelegt. An dieser Stelle sollen zunächst grundlegende Anforderungen an Operationen zur Manipulation von Sichtbarkeitsbereichen aufgestellt werden. Anschließend wird am Beispiel von TLMIN eine Umsetzung vorgestellt. Das verfolgte Ziel besteht darin, grundlegende Operationen zu identifizieren und zu implementieren, die die Manipulation von Sichtbarkeitsbereichen erlauben. Diese sollen so gewählt werden, daß sich aus ihnen höhere Strukturierungskonzepte aufbauen lassen, mit denen ähnliche wie die in Kapitel 3 beschriebenen Konzepte realisiert werden können. Im Gegensatz dazu erlaubt die Verwendung von primitiven, orthogonal kombinierbaren Konstrukten jedoch, die Strukturierungsmechanismen individuell anzupassen und neue, bislang unbekannte Mechanismen nachträglich (add on) einzuführen.

In diesem Abschnitt wird Bezug auf die bereits in Abschnitt 3.1 eingeführte Notation genommen. Sie dient auch in diesem Kapitel der Verdeutlichung von Sichtbarkeitskonstruktionen.

5.1.1 Anforderungen

Aus der Untersuchung von Sichtbarkeitsbereichsoperationen anderer Programmiersprachen in Kapitel 3 lassen sich fünf grundlegende Operationen, aus denen sich Sichtbarkeitszenarios aufbauen lassen, ableiten:

1. Anlegen eines Sichtbarkeitsbereichs. In der grafischen Notation entspricht dies einem neuen Kasten ohne obere Sichtbarkeitsgrenze. Diese Operation ist nur in Zusammenhang mit der unter Punkt 3 aufgeführten Operation sinnvoll. Sie markiert den Anfang eines Bereichs, in dem neu hinzukommende Bezeichner durch eine Sichtbarkeitsgrenze unsichtbar gemacht werden.

2. Anlegen eines Sichtbarkeitsbereichs bei gleichzeitiger Verdeckung aller bislang sichtbaren Bezeichner. Auch hierbei handelt es sich um eine Sichtbarkeitsgrenze, die bezogen auf die grafische Notation und im Unterschied zur vorstehend beschriebenen Operation eine Sichtbarkeitsgrenze an der oberen Kante eines Sichtbarkeitsbereichs aufweist.

3. Hinzufügen eines Bezeichners in einen Sichtbarkeitsbereich.

4. Beenden eines Sichtbarkeitsbereichs. Damit verbunden ist das Unsichtbarmachen aller in diesem Bereich hinzugefügten Bezeichner. Das Beenden eines Sichtbarkeitsbereichs entspricht der unteren Sichtbarkeitsgrenze (s. Abbildung 3.1).

5. Selektives Einführen von Bezeichnern aus einem anderen Bereich, nicht notwendigerweise den direkt umschließenden, in den aktuellen Sichtbarkeitsbereich.

Eine genaue Betrachtung der Operation 5 zeigt, daß diese nur in Zusammenhang mit dem Vorhandensein einer Sichtbarkeitsgrenze sinnvoll ist, da im anderen Fall bereits alle Bezeichner sichtbar sind. Weiterhin ist ersichtlich, daß Bezeichner nicht an einer Sichtbarkeitsgrenze am Ende eines Sichtbarkeitsbereichs eingeführt werden können. Die einzige sinnvolle Verwendung des selektiven Einfügens besteht darin, Bezeichner durch eine Sichtbarkeitsgrenze hindurch einzuführen. Da es sich bei TLMIN und bei allen anderen in Kapitel 3 betrachteten Programmiersprachen um blockstrukturierte Sprachen handelt, treten Operation 1 und 2 immer mit einer korrespondierenden Operation 4 auf.

Im folgenden werden die in TLMIN bereitstehenden Sprachkonstrukte zur Manipulation von Sichtbarkeitsbereichen untersucht. Dabei wird deutlich, daß jede Operation einem Sprachkonstrukt oder einer Kombination aus ihnen entspricht.

5.1.2 *begin … end* Block

Ein einfacher Fall ist der *begin … end* Block. Der am Anfang des Blocks gültige Sichtbarkeitsbereich entspricht dem des übergeordneten. Innerhalb des Blocks kann der Sichtbarkeitsbereich beliebig erweitert werden. Am Ende des Blocks wird der ursprüngliche Sichtbarkeitsbereich wiederhergestellt. Bereits im Bereich enthaltene Bezeichner bzw. deren Bindungen werden durch neue Bindungen innerhalb des *begin … end* Blocks, die unter gleichem Namen angelegt werden, überdeckt. Abbildung 5.1 verdeutlicht dieses Verhalten.

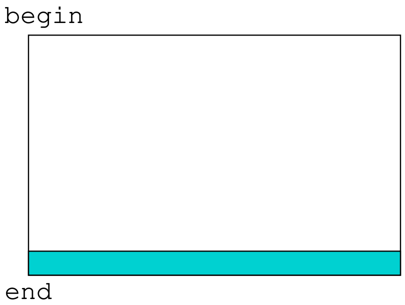


Abbildung 5.1: Sichtbarkeitsbereich eines *begin … end* Blocks.

Begin … end Blöcke treten in TLMIN in verschiedenen Kontexten und in einigen Varianten auf. Explizit ist die Verwendung eines *begin … end* Blocks bei Verwendung der Schlüsselwörter *begin* und *end* wie im nachstehenden Beispiel:

```
let a = begin
  let b = 42
  let c = 43
end
```

Die Bezeichner *b* und *c* sind nur innerhalb der *begin … end* Klammer sichtbar.

Sichtbarkeitsblöcke, wie in Abbildung 5.1 schematisch gezeigt, können jedoch auch ohne die Verwendung der Schlüsselwörter *begin* und *end* auftauchen, wie zum Beispiel bei Verwendung des folgenden Konstruktes:

```
if a == 47 then
  let b = 42
  let c = 43
else
  let b = 43
  let c = 42
end
```

In diesem Fall werden analog dem *begin … end* Fall zwei entsprechende Sichtbarkeitsbereiche, deren Grenzen durch *then … else* bzw. *else … end* markiert sind, aufgebaut. Ähnlich werden

Sichtbarkeitsbereiche in der Sprache TLMIN in Fallunterscheidungen (*case of*), Funktionsdefinitionen (Parametersignaturen), Tupeldefinitionen usw. gebildet.

In TLMIN existiert keine Operation, die es erlaubt, Bezeichner durch eine Sichtbarkeitsgrenze „nach unten“ zu exportieren. Dieses Verhalten kann jedoch mit den vorhandenen Mitteln simuliert werden.

```
let t = begin
   (* Berechnung der exportierten Bezeichner a, b und c *)
  tuple
    let a = ...
    let b = ...
    let c = ...
  end
end
open t
```

Ähnlich wie in der gezeigten Sequenz läßt sich dieses Verhalten durch Bindung eines globalen Bezeichners innerhalb des Sichtbarkeitsblockes an ein lokales Objekt erreichen. Unabhängig von der verwendeten Technik, bleibt festzuhalten, daß Bezeichner innerhalb eines Sichtbarkeitsblockes nur mit einem global existierenden Namen über die Sichtbarkeitsgrenze hinaus sichtbar gemacht werden können.

5.1.3 Funktion

Der Sichtbarkeitsbereich einer Funktion setzt sich, bedingt durch die Parameterübergabe an die Funktion, aus zwei Blöcken zusammen (s. Abbildung 5.2).

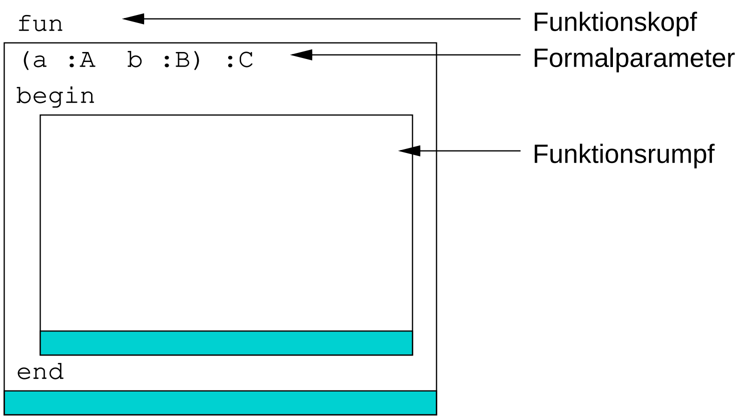


Abbildung 5.2: Sichtbarkeitsbereich einer Funktion.

- Der Rumpf der Funktion wird durch einen *begin* ... *end* Block gebildet. In der Sprache TLMIN kann dies durch die Verwendung der Schlüsselwörter *begin* und *end* deutlich gemacht werden. Das bedeutet auch, daß innerhalb des Funktionsrumpfes alle außerhalb definierten Bezeichner sichtbar sind. Die Schlüsselwörter *begin* und *end* können weggelassen werden, wenn der Funktionsrumpf nur aus einer (geschachtelten) Anweisung besteht.

- Der äußere Sichtbarkeitsbereich sorgt dafür, daß die Parameter in den inneren, den Sichtbarkeitsbereich des Rumpfes gelangen, aber nicht außerhalb der Funktion definiert sind. Die Namen bzw. Bindungen der Parameter werden an die Aktualparameter gebunden, die aus dem Sichtbarkeitsbereich des Funktionsaufrufenden stammen. Diese dynamische Zuweisung an die Aktualparameter ist in Abbildung 5.2 durch die Notation nicht explizit dargestellt.

Beachtenswert ist an dieser Stelle, daß sich im Sichtbarkeitsbereich der Funktion alle zur Definitionszeit existierenden Bezeichner befinden, die somit auch zur Ausführungszeit referenzierbar sind. Die Menge der tatsächlich referenzierten Bezeichner bezeichnet man auch als Funktionsabschluß (*function closure*).

Abbildung 5.2 zeigt weiterhin, daß der innere Sichtbarkeitsbereich keine Auswirkung auf die gesamte Sichtbarkeitssituation hat, da nach Abschluß des inneren und des äußeren Bereichs keine Anweisungen stehen können. Er könnte somit entfallen, macht jedoch die durch die Schlüsselwörter *begin* und *end* gebildete Klammer deutlich und ist deshalb in der Darstellung enthalten.

5.1.4 Abgeschlossener Sichtbarkeitsbereich

Sichtbarkeitsbereiche gemäß Operation vier aus Abschnitt 5.1.1 werden in TLMIN durch die folgenden beiden Konstrukte angelegt:

scope [with . . . do] end

für Werte und

Scope [with . . . do] end

für Typen.

Über die optionale Verwendung von *with . . . do* gestatten sie sowohl das Einschränken von Sichtbarkeitsbereichen an deren Beginn als auch das selektive Einfügen. Ein Konstrukt bezieht sich auf die Einschränkung der Sichtbarkeitsbereiche von Werten, das andere Konstrukt auf die Sichtbarkeitsbereiche von Typen. Die Unterschiede in der Semantik sind:

- Das Sichtbarkeitskonstrukt für Werte liefert einen Wert zurück. Es wird während des Typüberprüfens auf einen Sequenzblock abgebildet, liefert also den Wert der letzten Bindung zurück.

- Das Sichtbarkeitskonstrukt für Typen liefert einen Typ zurück. Bei korrekter Typisierung wird der im Sichtbarkeitsblock eingeschlossene Typ zurückgegeben.

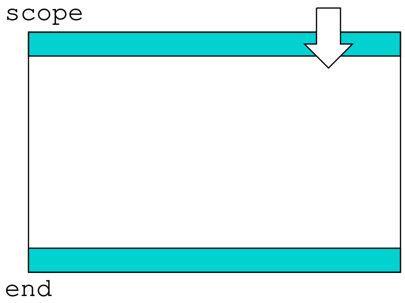


Abbildung 5.3: Abgeschlossener Sichtbarkeitsbereich.

Abbildung 5.3 zeigt einen abgeschlossenen Sichtbarkeitsbereich. Die folgende kurze Sequenz von Anweisungen verdeutlicht die Verwendung der Sichtbarkeitsoperation. Sie führt zu einer Fehlermeldung, da sich der Bezeichner *a* nicht im Sichtbarkeitsbereich, der durch *scope* ... *end* angelegt wird, befindet.

```
let a = 3
let b = scope a + 3 end
⇒ Unbound identifier a.
```

Abweichend dazu führt die nachfolgende Anweisungsfolge nicht zu einer Fehlermeldung des Compilers, da der Bezeichner *a* explizit in den Sichtbarkeitsbereich eingeführt wird.

```
let a = 3
let b = scope with a infix+ do a + 3 end
⇒ 6 :Int
```

Für das Sichtbarkeitskonstrukt ist keine Laufzeitunterstützung notwendig. Das ist begründet in der Tatsache, daß es sich hierbei um eine rein statische Abprüfung von Sichtbarkeit bzw. Vorhandensein von Bezeichnern handelt, die zur Übersetzungszeit vollständig behandelt werden kann.

Initiale Bezeichner

In einem neuen Sichtbarkeitsbereich, der durch *scope*... *end* erzeugt wird, sind per Definition keine Bezeichner sichtbar. Hierunter fallen auch initiale Bezeichner wie elementare Datentypen (z.B. *Bool*, *Int* usw.) und elementare Operationen auf diesen (Addition, Subtraktion usw.), die bereits im Compiler vorhanden sind und nur durch Einrichten eines entsprechenden Verweises in die Umgebung sichtbar gemacht werden müssen.

Hier stellt sich nun die generelle Frage, ob es einen Satz vordefinierter Bezeichner geben soll, der zu jeder Zeit sichtbar ist und der auch durch Sichtbarkeitsgrenzen nicht unsichtbar gemacht werden kann. Das ist z.B. bei Modulen in Modula-2 zu beobachten, wo trotz einer Sichtbarkeitsgrenze am Beginn eines Moduls initiale Bezeichner sichtbar sind. Weiterhin stellt sich die Frage, welche Bezeichner in diese Kategorie fallen. In der Implementierung von Modula-2 sind alle initialen Bezeichner in allen Sichtbarkeitsbereichen des Systems verfügbar.

Die in der Sprache TLMIN gewählte Lösung sieht vor, die initialen Bezeichner in einem eigenen Modul zu kapseln. In jedem abgetrennten Sichtbarkeitsbereich, in dem diese initialen Bezeichner Verwendung finden, müssen sie importiert werden. Der Vorteil dieser Lösung liegt zum einen darin, daß das Konzept der Sichtbarkeitsgrenzen semantisch rein definiert ist. Eine Sichtbarkeitsgrenze entspricht dem Namen Grenze, da tatsächlich keine Bezeichner „hindurchkommen“. Zum anderen kann die Diskussion, welche Bezeichner initial sind, dem Anwender überlassen werden, indem er die Freiheit erhält, das Modul der initialen Bezeichner selbst zu gestalten bzw. unterschiedliche zu definieren.

5.1.5 Selektiv eingeführte Bezeichner

Beim Konstrukt zum Konstruieren eines abgeschlossenen Sichtbarkeitsbereichs ist bereits eine Möglichkeit zum Importieren von Bezeichnern vorgesehen. Darüber hinaus kann es sinnvoll sein, Bezeichner innerhalb eines Blockes einzuführen und diese nicht schon zu Beginn sichtbar zu machen. In TLMIN kann dies mit der *use* Klausel formuliert werden.

```
let a = 3
let b = scope
  let c = 42
  use a
  let d = a * c
end
⇒ 126 :Int
```

Im vorstehenden Beispiel wird der Bezeichner *a* erst nach der Bindung von *c* aus dem umschliessenden Sichtbarkeitsbereich eingeführt. Das *use* Konstrukt ist die Implementierung der fünften elementaren Operation auf Sichtbarkeitsbereichen. Abbildung 5.4 zeigt die Umsetzung in der grafischen Notation.

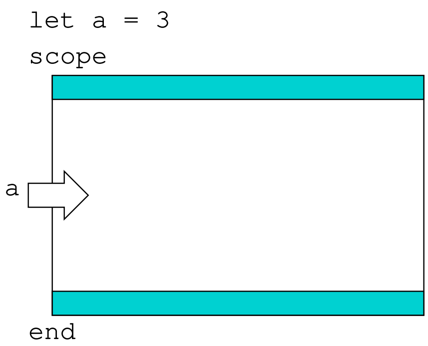


Abbildung 5.4: Selektiv eingeführter Bezeichner.

Die Umsetzung in TLMIN ist so realisiert, daß sich die *use* Anweisung jeweils auf die letzte Sichtbarkeitsgrenze bezieht. Es ist z.B. nicht sinnvoll, Bezeichner in einen *begin* … *end* Block einzuführen, da diese dort ohnehin sichtbar sind. Die *use* Anweisung bezieht sich damit nicht auf den direkt umschließenden Sichtbarkeitsbereich, sondern präziser auf den durch eine Sichtbarkeitsgrenze abgeschlossenen umschließenden Sichtbarkeitsbereich.

5.2 Höhere Strukturierungskonzepte

In diesem Abschnitt wird gezeigt, wie man mit Hilfe der bereits eingeführten elementaren Sprachkonstrukte zur Manipulation von Sichtbarkeitsbereichen höhere Strukturierungskonzepte realisieren kann. Als Beispiel werden Module und Schnittstellen formuliert, die in modulorientierten Programmiersprachen wie z.B. Modula-2 und TL zur Systemstrukturierung eingesetzt werden. Diese höheren Konzepte dienen dazu, die unabhängige Entwicklung und Implementierung und damit die in Kapitel 1 aufgeführten Prinzipien zu unterstützen. Dazu müssen diese höheren Strukturierungseinheiten separat übersetzbar sein, d.h. der Entwurf und die Implementierung eines Moduls kann zeitlich schon dann erfolgen, wenn von den benutzten (importierten) Modulen nur die Schnittstellen und nicht die Implementierungen vorliegen. Zum einen wird damit die Anwendung der Prinzipien Modularisierung und Wiederverwendbarkeit unterstützt, da die Module und Schnittstellen ein Programm bereits in kleinere Einheiten zerlegen. Module als abgeschlossene Einheiten können in anderen Programmen wiederverwendet werden, alle abhängigen Teile sind in der Importliste festgeschrieben. Zum anderen wird durch die Aufteilung in Module und Schnittstellen die Entwicklung in einem Programmierteam unterstützt, da die beteiligten Entwickler nach Festlegung der Schnittstellen unabhängig voneinander Module entwerfen, ausprogrammieren und testen können.

Als weitere Anforderungen an Module und Schnittstellen werden die folgenden Punkte aufgeführt.

- Module können mehreren Schnittstellen genügen, wenn die Schnittstellen in einer Subtypbeziehung zueinander stehen.
- Eine Schnittstelle kann von mehreren dem Typ der Schnittstelle entsprechenden Modulen implementiert werden.
- Module sind flexibel konfigurierbar. D.h. bei der Instantiierung eines Moduls, also zum Zeitpunkt des Bindens, besitzt der Programmierer die Freiheit, das Modul mit beliebigen, dem Typ der importierten Bezeichner entsprechenden Einheiten zu parametrisieren.
- Schnittstellen sind wie Module flexibel konfigurierbar.

Die Module und Schnittstellen können über eine Syntaxerweiterung dem Programmierer über Schlüsselworte zur Verfügung gestellt werden. Durch diesen Ansatz ist es möglich, auch andere Konzepte wie z.B. Klassen in die Sprache einzuführen, ohne jedoch die eigentliche Kernsprache TLMIN zu verändern und den Compiler modifizieren zu müssen. Lediglich der Parser wird mit der neuen Grammatik parametrisiert. In bestimmten Situationen muß der Übersetzungsvorgang der Softwareeinheiten werkzeugunterstützt erfolgen. Wie in Abschnitt 4.4 gezeigt wurde, ist hierzu jedoch kein Eingriff in den Übersetzer selbst notwendig. Die erforderlichen Anpassungen können komplett auf der Anwendungsebene erfolgen.

Als Vorteil dieses Ansatzes ist auch zu sehen, daß alle compilerinternen Repräsentationen und Algorithmen nicht nur unverändert bleiben, sondern auch weiterhin kompatibel sind, d.h., daß Objekte, die mit einem um Klassen erweiterten TLMIN Übersetzer erzeugt wurden, kompatibel zu Objekten eines um Module und Schnittstellen erweiterten Compilers sind. Programmierer können also nach eigenen Präferenzen Systemstrukturierungskonzepte auswählen und dennoch Code und Zwischenrepräsentationen des Übersetzers austauschen.

5.2.1 Schnittstellen

Schnittstellen werden zwischen Programmen und zwischen abgeschlossenen Programmteilen (Modulen) definiert [Duden 1988]. Dazu formuliert man alle von einem Modul auszuführenden Aufgaben, die erforderlichen Attribute, die bereitgestellten Funktionen einschließlich Parameter, die Zugriffsmöglichkeiten auf Daten und die Beziehungen zwischen ihnen [Duden 1988]. Ergänzend dazu kommen Angaben zur Reihenfolge der Benutzung sowie Vor- bzw. Nachbedingungen. Softwaretechnisch ermöglichen Schnittstellen den unabhängigen Entwurf und die unabhängige Implementierung von Programmeinheiten dadurch, daß sie den importierenden Modulen und Schnittstellen die gewünschten Bezeichner mit Information über ihren Typ zur Verfügung stellen und sichtbar machen.

Bezogen auf TLMIN müssen Schnittstellen Module spezifizieren, indem sie die Signaturen aller exportierten von außen nutzbaren Funktionen enthalten. Weiterhin werden Typen und Werte, die ein Modul exportiert, in der Schnittstelle aufgeführt.

```
Let Stack =  Scope with StdType StdIde do
Oper(Let StdType = StdType()
     Let StdIde = StdIde(:StdType)  stdIde :StdIde)
  Scope with stdIde do
    Tuple
      T <: Oper(E <:Ok) Ok
      new (E <:Ok) :T(:E)
      empty (E <:Ok stack :T(:E)) :stdIde.Bool
      push (E <:Ok element :E stack :T(:E)) :T(:E)
      pop (E <:Ok stack :T(:E)) :T(:E)
      top (E <:Ok stack :T(:E)) :E
    end
  end
end
```

Abbildung 5.5: Schnittstellendefinition für ein Stack-Modul mit Hilfe von TLMIN Basisprimitiven.

Abbildung 5.5 zeigt die Definition einer Schnittstelle, die ein Stack-Modul spezifiziert. Mit Hilfe eines Tupeltypen werden die exportierten Bezeichner aggregiert. Sie stellen die eigentliche

Schnittstelle dar. Innerhalb dieses Tupels werden nun die zuvor importierten Bezeichner benutzt. Hierbei kann es sich sowohl um Schnittstellen als auch Module handeln, wobei bei letzteren nicht zugesichert werden kann, daß diese zum Zeitpunkt der Übersetzung dieser Schnittstelle existieren. Die angestrebte separate Übersetzung läßt es sogar explizit zu, daß diese Module nicht existieren müssen. Aus diesem Grund wird die Auswertung dieses Tupeltypen durch einen Typoperator verzögert. Er wird mit den importierten Bezeichnern parametrisiert, die in der Parameterliste lokal gebunden werden. Sie müssen dazu jedoch mit den ihrerseits importierten Bezeichnern parametrisiert werden. Letztendlich werden in der Parameterliste des Typoperatoren somit alle transitiv erreichbaren Bezeichner aufgeführt.

Die Sichtbarkeit des Schnittstellenrumpfes wird durch zwei Scope-Konstrukte determiniert, Abbildung 5.6 verdeutlicht dies anhand der eingeführten Notation. Diese stellen folgendes sicher:

- Das äußere Scope-Konstrukt sorgt dafür, daß alle Typen, die direkt oder indirekt (transitiv) importiert werden, in den Sichtbarkeitsbereich der Typoperatordefinition gelangen. Modulbezeichner werden nicht zum Zeitpunkt der Definition in den Sichtbarkeitsbereich des Typoperators gebracht, sondern bei der Instantiierung als Parameter übergeben[1]. Das Beispiel in Abschnitt 5.2.4 macht dies deutlich.

- Das innere Scope-Konstrukt bringt diejenigen Bezeichner in den Sichtbarkeitsbereich des Schnittstellenrumpfes, die dort tatsächlich referenziert werden. Bei der Formulierung mit Hilfe einer Syntaxerweiterung wie in Abbildung 5.8 sind dies die in der Import-Klausel aufgeführten Bezeichner. Hier treten nun sowohl Typ- als auch Wertbezeichner auf.

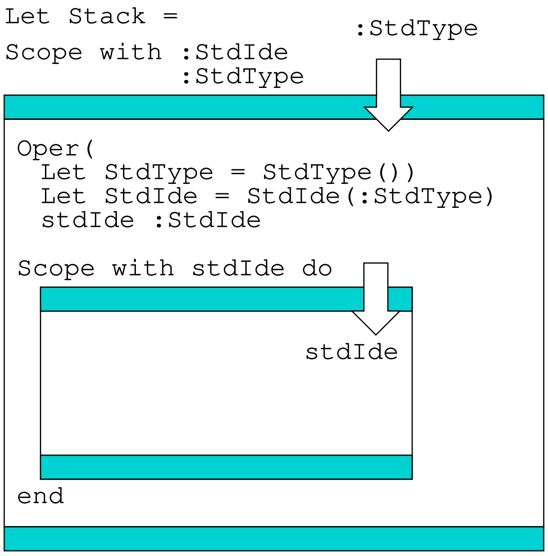


Abbildung 5.6: Sichtbarkeitsszenario einer Schnittstelle.

[1] Im Gegensatz zu TL ist es in TLMIN möglich, Typoperatoren mit Wertparametern zu formulieren und zu instantiieren. Dies ist eine Voraussetzung, um diese Abbildung realisieren zu können.

Alle in der Schnittstelle verwendeten, jedoch nicht importierten, Bezeichner führen dazu, daß die Schnittstellendefinition vom Typüberprüfer zurückgewiesen wird. Dies ist insbesondere durch das innere Scope-Konstrukt gewährleistet, dessen Abwesenheit es ermöglichen würde, alle auch transitiv erreichbaren Bezeichner zu referenzieren, ohne diese in der Import-Klausel aufzuführen. Die Scope-Konstrukte dienen damit der Unterscheidung von Bezeichnern in zwei Gruppen:

- Bezeichner, die zum Instantiieren der Schnittstelle (Typoperator) notwendig sind.
- Bezeichner, die zum Typüberprüfen des Schnittstellenrumpfes notwendig sind.

Da eine Schnittstelle Typinformation repräsentiert und Module Werte dieses Typs darstellen (s. Abschnitt 5.2.2), können zu einer Schnittstelle verschiedene Modulausprägungen existieren, so wie zu einem Typen mehrere Werte existieren können.

```
Let Stack2 = Scope with StdType do
Oper(Let StdType = StdType())
     Let StdIde = StdIde(:StdType)  stdIde :StdIde)
  Scope with stdIde do
    Tuple
      T <: Oper(E <:Ok) Ok
      new (E <:Ok) :T(:E)
      empty (E <:Ok stack :T(:E)) :stdIde.Bool
      push (E <:Ok element :E stack :T(:E)) :T(:E)
      pop (E <:Ok stack :T(:E)) :T(:E)
    end
  end
end
```

Abbildung 5.7: Generalisierte Schnittstellendefinition für ein Stack-Modul.

Schnittstellendefinitionen bzw. die ausgewerteten Typoperatoren können gemäß der Subtypregeln für Tupeltypen in Beziehung zueinander stehen. So ist z.B. die in Abbildung 5.7 dargestellte Schnittstelle *Stack2* ein Supertyp der Schnittstelle *Stack*, da sie weniger Komponenten enthält, die verbleibenden aber denen aus *Stack* entsprechen. Somit können Module der Schnittstelle *Stack* dort auftreten, wo Module der Schnittstelle *Stack2* gefordert sind, wobei durch den Übersetzer nur die syntaktische und typmäßige Korrektheit zugesichert wird. Zusicherungen, daß ein Modul auch von der implementierten Funktionalität her der Schnittstelle bzw. den Erwartungen an sie entsprechen, können nicht gegeben werden. Allgemein ist die Generalisierung von Schnittstellen nicht auf das Weglassen von Bezeichnern bzw. Signaturen beschränkt. Es kommen vielmehr die in [Matthes 1993] und [Bremer 1996] ausgeführten Typregeln für Tupeltypen zur Anwendung.

Schnittstellen beziehen sich bei ihrer Definition auf andere Bezeichner (Import). Von diesen Bezeichnern wird ausschließlich die Typinformation genutzt. Es reicht somit aus, Schnittstellen und Module mit der Typinformation importierter Bezeichner zu versorgen. Diese steht jedoch nicht ausschließlich in anderen Schnittstellen, sondern im Falle von abstrakten Datentypen in den Modulen, die Datentyp und Funktionen auf Objekten dieses Typs aggregieren und die in der Schnittstelle die Typinformation nicht offenlegen.

Die Schnittstellen repräsentierenden Typoperatoren müssen deshalb explizit mit den Modulwerten der importierten Bezeichner instantiiert werden. Nur auf diese Weise ist eine Kompatibilität von Objekten abstrakter Datentypen gewährleistet.

```
Let Stack = Stack(:StdType :StdIde stdIde);
```

Dieser Aspekt wird in Abschnitt 5.2.3 wieder aufgegriffen.

```
interface Stack
import
  stdIde
export
  T <: Oper(E <:Ok) Ok
  new (E <:Ok) :T(:E)
  empty (E <:Ok stack :T(:E)) :stdIde.Bool
  push (E <:Ok element :E stack :T(:E)) :T(:E)
  pop (E <:Ok stack :T(:E)) :T(:E)
  top (E <:Ok stack :T(:E)) :E
end
```

Abbildung 5.8: Schnittstellendefinition für ein Stack-Modul mit Hilfe von Syntaxerweiterung.

Durch die Verwendung des Scope-Konstruktes sind alle Bezeichner, die nicht in der dem Schlüsselwort *with* folgenden Liste aufgeführt sind, verdeckt. Dies gilt auch für die Standardtypen und -operationen, wie Int, Bool usw. Sollen diese in der von Sprachen wie Modula-2 gewohnten Weise zur Verfügung stehen, ist es notwendig, die in den Schnittstellen *StdType* und *StdIde* befindlichen Bezeichner explizit sichtbar zu machen. [2]

Um dem Programmierer die fehlerträchtige Schreibweise, wie sie in Abbildung 5.5 und 5.7 verwendet wird, zu ersparen, kann mittels Syntaxerweiterung eine intuitive Schreibweise gewählt werden. Ein Beispiel hierfür ist in Abbildung 5.8 gegeben. Die Darstellung in Abbildung 5.5 kann jedoch nicht durch rein textuelle Umformungen aus der Formulierung in Abbildung 5.8 gewonnen werden, da die Information über transitiv importierte Bezeichner aus der Darstellung in Abbildung 5.8 nicht hervorgeht. Diese muß durch ein Werkzeug bereitgestellt werden (s. Abschnitt 4.4), das die Abhängigkeitsinformation während des Übersetzungsvorganges, meist während des Parsens, sammelt.

[2] Das führt jedoch dazu, daß über diese Schnittstelle alle diese Bezeichner exportiert werden und auch im entsprechenden Modul aufgeführt werden müssen. Die Schnittstelle wird dadurch nicht nur besonders groß, sondern führt zusätzliche Bezeichner für die Standardtypen ein, wie z.B. *Stack.int_T* durch die Schnittstelle *Stack*. Der Typüberprüfer erkennt jedoch die Gleichheit aller auf diese Weise erzeugten Typen. So ist *Stack.int_T* gleich *Stack2.int_T* und diese gleich *stdType.Int*.

5.2.2 Module

Ein Modul ist nach [Duden 1988] die Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren zu einer Einheit. Soll ein Modul von einem anderen benutzt werden, so muß angegeben werden, welche Teile des Moduls von außen sichtbar sein sollen und welche nicht. Dies festzulegen ist Aufgabe der Schnittstellen (s. Abschnitt 5.2.1). Die konkrete Realisierung insbesondere der Datentypen und der Prozeduren soll jedoch verborgen bleiben. Daraus ergeben sich Anforderungen an die Sichtbarkeitsszenarios von Modulen:

```
let stackLINK =
scope with List StdType Stack StdIde do
  fun(Let StdType = StdType()  stdType :StdType
      Let StdIde = StdIde(:StdType)  stdIde :StdIde
      Let List = List(:StdType :StdIde stdIde)  list :List
      Let Stack = Stack(:StdType :StdIde stdIde)) :Stack
  scope with list stdType Stack do
    Let T(E <:Ok) = list.T(:E)
    let new (E <:Ok) :T(:E) =
      list.new (:E)
    let empty (E <:Ok stack :T(:E)) :stdIde.Bool =
      list.empty (stack)
    let push (E <:Ok element :E stack :T(:E)) :T(:E) =
      list.cons (element stack)
    let pop (E <:Ok stack :T(:E)) :T(:E) =
      list.tail (stack)
    let top (E <:Ok stack :T(:E)) :E =
      list.head (stack)
    let result :Stack = tuple
      Let T = T
      let new = new
      let empty = empty
      let push = push
      let pop = pop
      let top = top
    end
    result
  end
end
```

Abbildung 5.9: Modulimplementierung gemäß der Schnittstelle *Stack*.

- Ein Modul kann ein anderes benutzen und muß folglich Zugriff auf entsprechende Typ- und Wertinformation haben.
- Die Implementationen von Datentypen und Prozeduren sind gemäß der Schnittstelle sichtbar; d.h., daß Typen, die in der Schnittstelle abstrakt definiert wurden, durch das Modul, in dem eine konkrete Ausformulierung erfolgen muß, nicht ihre Implementierungsdetails preisgeben dürfen. Die Sichtbarkeitsregeln für Funktionen sorgen bereits dafür, daß im Rumpf definierte Bezeichner nicht nach außen sichtbar werden (s. Abschnitt 5.1.3).
- Alle nicht in der Schnittstelle aufgeführten Bezeichner sind außerhalb des Moduls nicht sichtbar.

Darüber hinaus muß ein Modul, wie in Abschnitt 5.2 ausgeführt, separat übersetzbar sein, um unabhängige Entwicklung und Implementierung zu unterstützen.

Ein Modul soll nach Möglichkeit mehreren Schnittstellen genügen können, z.B. wenn die Schnittstellen unterschiedlich viele Bezeichner exportieren, wie die Schnittstellen *Stack* und *Stack2*. Das Modul *stack* entspricht nach den Subtypregeln beiden Schnittstellen. Der von der entsprechenden Funktion evaluierte Wert, das Modul, genügt sowohl *Stack* als auch *Stack2*. Um dies zu erreichen, wird hier der maximale Typ angegeben, also der Typ, der die spezialisiertesten Angaben macht (*Stack*). Alle Schnittstellen, denen das Modul ebenfalls genügt, sind Supertypen dieses Typs.

```
module stack :Stack
import
  list
  stdIde
export
  Let T(E <:Ok) = list.T(:E)
  let new (E <:Ok) :T(E:) =
    list.new (:E)
  let empty (E <:Ok stack :T(:E)) :stdIde.Bool =
    list.empty (stack)
  let push (E <:Ok element :E stack :T(:E)) :T(:E) =
    list.cons (element stack)
  let pop (E <:Ok stack :T(:E)) :T(:E) =
    list.tail (stack)
  let top (E <:Ok stack :T(:E)) :E =
    list.head (stack)
end
```

Abbildung 5.10: Modulformulierung mittels Syntaxerweiterung.

In TLMIN kann ein Modul wie in Abbildung 5.9 formuliert werden. Analog zu Schnittstellen wird die eigentlich gewünschte Information, das Modul, mit Hilfe eines Tupels aggregiert. Bei einem Modul verfolgt man mit der expliziten Bindung an ein Tupel weiterhin, daß die Reihenfolge der zuvor aufgeführten Bezeichner keinen Einfluß auf den Modulwert nimmt. Durch die Verwendung von Schnittstelle und Modulen in der Importliste entsteht auch bei einem Modul die Notwendigkeit, die Auswertung des Moduls (die Evaluierung des Tupelwertes) zu verzögern, weil die referenzierten Module zum Definitionszeitpunkt noch nicht vorliegen müssen. Es wird bei der Formulierung eines Moduls ein ähnliches Konstrukt wie bei Schnittstellen in Abschnitt 5.2.1 benutzt, da es sich bei dem Modul jedoch um einen Wert handelt, wird zur Verzögerung der Auswertung eine Funktion verwendet.

Wie bei Schnittstellen bedienen sich Module ebenfalls zweier geschachtelter Scope-Konstrukte. Auch hier verfolgen sie den Zweck, einerseits die transitiv erreichbaren Schnittstellen in den Sichtbarkeitsbereich der Funktion zu bringen und andererseits diejenigen Bezeichner, die innerhalb des Moduls tatsächlich referenziert werden, für den Modulrumpf bereitzustellen. Innerhalb des Funktionskopfes werden sowohl die direkt als auch die indirekt importierten Schnittstellen und Module angegeben. Die transitiv erreichbaren Typen müssen in der Funktionssignatur angegeben werden, da diese zum Definitionszeitpunkt des Moduls noch nicht instantiiert sein müssen. Durch die besondere Schreibweise der Typparameter bzw. deren Signaturen sind diese für nachfolgend aufgeführte Signaturen sichbar. So sind z.B. in Abbildung 5.9 die Typbezeichner *StdType* und *StdIde* sowie der Wert *stdIde* für den Ausdruck *Stack(:StdType :StdIde stdIde)* sichtbar.

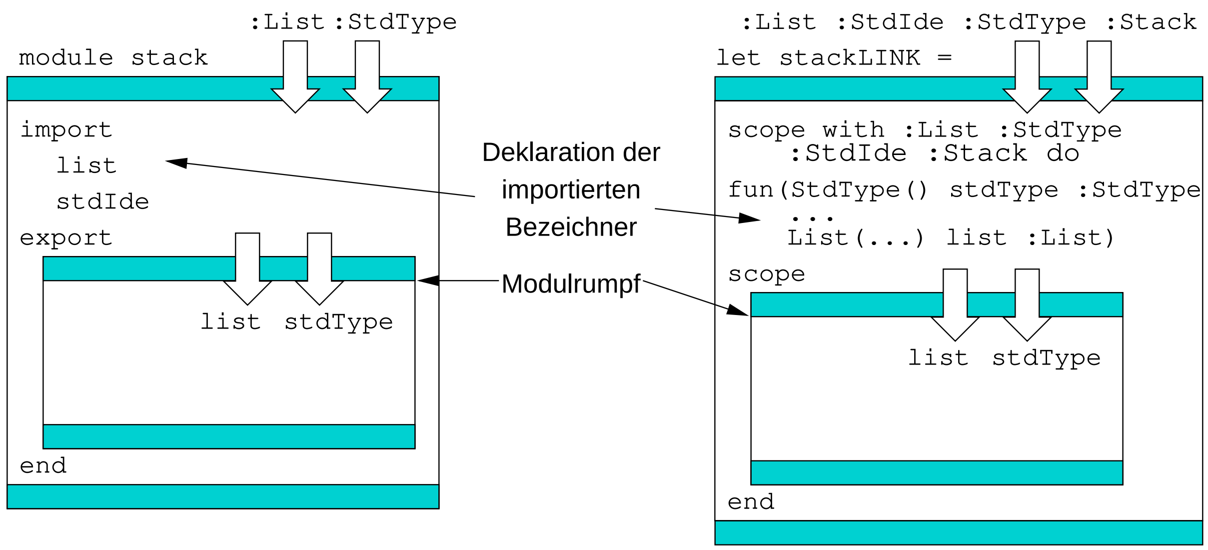


Abbildung 5.11: Sichtbarkeitsbereich eines Moduls.

Um beim Bindevorgang nicht auf Module angewiesen zu sein, die exakt dem in der Importliste angegebenen Namen entsprechen, hat die Link-Funktion *stackLINK* Parameter gemäß ihrer transitiven Importe. Alternativ hätten die Namen nur in der Scope-Anweisung auftreten können, und die Funktion *stackLINK* an sich wäre parameterlos gewesen. Das Problem, daß abstrakte Datentypen genau über den Pfad angesprochen werden müssen, über den sie definiert sind, könnte damit nicht gelöst werden. Konkret heißt das, daß bei der Verwendung eines abstrakten Datentyps, der in einem Modul definiert ist, der Modulbezeichner angegeben werden muß. Auch das ist durch die Verwendung von Funktionsparametern gewährleistet.

Damit die Reihenfolge der Funktionsdefinitionen nicht der Reihenfolge der Schnittstellenbeschreibung entsprechen muß, sondern dem Programmierer größtmögliche Freiheit gegeben werden kann, wird der Rückgabewert (*result*) der Funktion *stackLINK* explizit gebunden.

Bei der Formulierung von Modulen wird die Möglichkeit der expliziten Einschränkung des Sichtbarkeitsbereichs genutzt, um den Funktionsabschluß auf die tatsächlich importierten Bezeichner zu beschränken. Durch die Beschränkung auf Typen im äußersten Scope-Konstrukt kann ein Modul separat, d.h. ohne das Vorhandensein anderer Module, übersetzt werden. Weiterhin wird eine Funktion eingesetzt, die eine verzögerte Bindung an importierte Module erlaubt. Abbildung 5.11 zeigt die grafische Repräsentation der Sichtbarkeitsbereiche eines Moduls. Dabei wird die Formulierung mittels Syntaxerweiterung der Beschreibung mit TLMIN Sprachprimitiven gegenübergestellt.

5.2.3 Binden

Das Modell der in Abschnitt 5.2.1 und 5.2.2 vorgestellten Schnittstellen und Module sieht vor, daß sowohl die Schnittstellen als auch die Module instantiiert werden müssen, wenn die referenzierten (importierten) Bezeichner gebunden sind, um zum Zeitpunkt der semantischen Analyse nicht auf die Anwesenheit der Modulwerte angewiesen zu sein. Genau dies ist die Bedingung für separates Übersetzen. Für Schnittstellen entspricht diese Instantiierung der Anwendung eines Typoperators. Dies ist im folgenden für die in Abschnitt 5.2.1 eingeführten Schnittstellen *Stack* und *Stack2* gezeigt:

```
Let Stack = Stack(:StdType :StdIde stdIde);
Let Stack2 = Stack2(:StdType :StdIde stdIde);
```

Die in Abbildung 5.9 dargestellte Funktion liefert mit jedem Aufruf einen Wert des angegebenen Typen *Stack* und damit bei jedem Aufruf ein Modul der Schnittstelle *Stack*. Beachtenswert ist hier, daß die Funktion *stackLINK* erst dann ausgeführt werden kann, wenn alle transitiv importierten Schnittstellen und Module instantiiert wurden. Die Auswertung der Schnittstelle *Stack* zum entsprechenden Typ muß also vorher geschehen[3]. Durch die folgenden zwei Aufrufe werden zwei Module evaluiert.

```
let stackA :Stack = stackLINK(:StdType stdType
                              :StdIde stdIde
                              :List list :Stack);
let stackB :Stack = stackLINK(:StdType stdType
```

[3]Die Instantiierung der Schnittstellen *List* und *StdType* und der Module *list* und *stdType* wird hier nicht vorgeführt, erfolgt aber analog zum Gezeigten. An dieser Stelle wird ebenfalls darauf verzichtet, die Schnittstellen *List* und *StdType* als Typoperatoren mit entsprechenden Parametern zu zeigen, weil dies die Übersichtlichkeit negativ beeinflußt hätte, ohne zum Verständnis beizutragen.

```
                                 :StdIde stdIde
                                 :List list :Stack);
```

Eine Schnittstelle spezifiziert auf diese Weise mehrere Module. Sie kann unterschiedlich implementiert werden.

Durch die Typregeln von TLMIN kann ein durch *stackLINK* erzeugtes Modul an einen Bezeichner, der einem Supertyp des eigenen Typs (maximale Spezifikation von *Stack*) genügt, gebunden werden.

```
let stack :Stack = stackLINK(:StdType stdType
                             :StdIde stdIde
                             :List list :Stack);
let stack2 :Stack2 = stackLINK(:StdType stdType
                             :StdIde stdIde
                             :List list :Stack);
```

Das Modul *stack2* besitzt keine sichtbare Komponente *top*. Diese wird durch die obige Zuweisung ausgeblendet. Modulinterne Zugriffe sind jedoch weiterhin möglich, da hier nur die Sichtbarkeit nach außen eingeschränkt wird. Das Beispiel zeigt, daß ein Modul mehrere Schnittstellen implementieren kann.

Die Link-Funktion muß mit den importierten Bezeichnern, also anderen Modulen und Schnittstellen, parametrisiert werden. Sie werden bei der Evaluierung der Funktion als Aktualparameter der Funktion *stackLINK* übergeben. Dadurch muß der Name, der innerhalb des Moduls *stack* für einen importierten Bezeichner benutzt wird, nicht dem tatsächlichen Namen entsprechen.

Wird in dem bisher verfolgten Beispiel die Implementation des importierten Moduls *list* vom Typ *:List* geändert, so kann das Modul *stack*, repräsentiert durch *stackLINK*, ohne neu übersetzt zu werden, an ein anderes Modul gleichen Typs bzw. eines Subtyps gebunden werden.

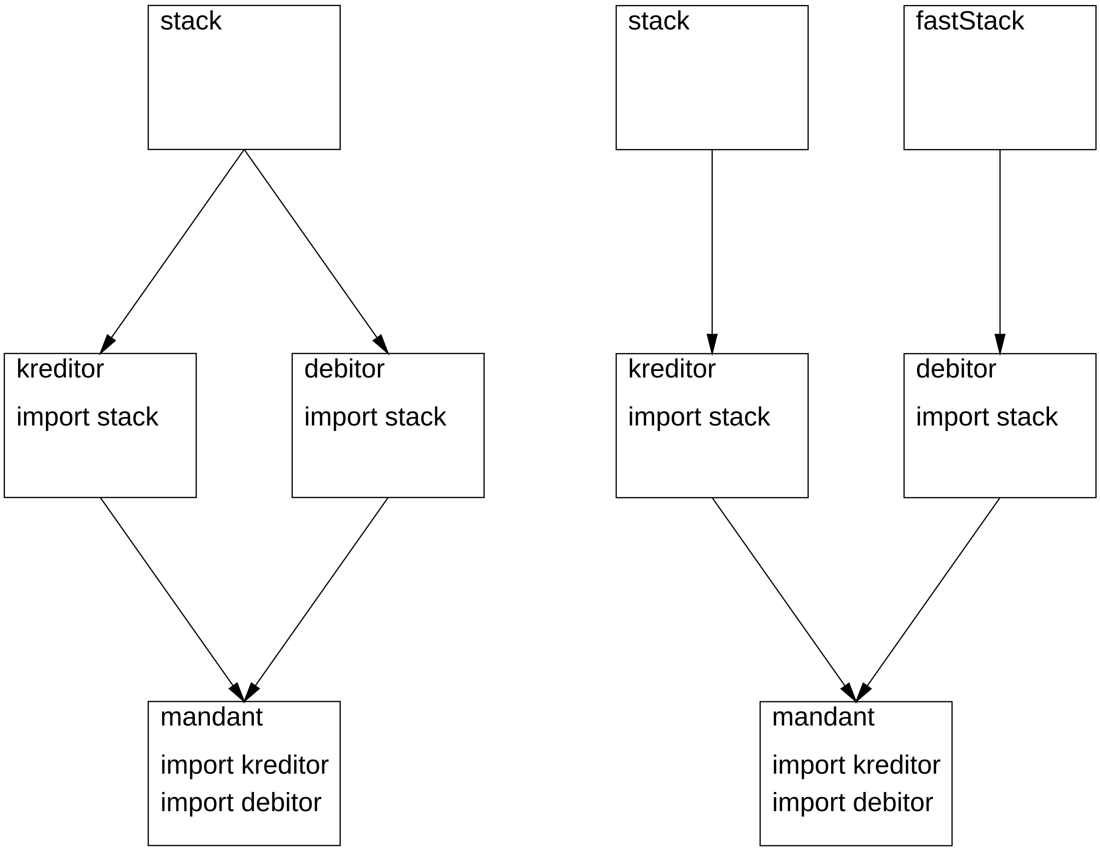
```
let fastStack :Stack = stackLINK(:StdType stdType
                                 :StdIde stdIde
                                 :List fastList :Stack);
```

Ein Modul kann an unterschiedliche Module gebunden werden, solange sie dem angegebenen Typ genügen. So kann das Modul *fastList* mehr Funktionen als das Modul *list* aufweisen, es muß jedoch in einer Subtypbeziehung zu diesem stehen. Zu beachten ist allerdings, daß durch die Verwendung unterschiedlicher Implementationen für die Komponente *list* des Moduls *fastStack* Objekte des Typs *fastStack.T* nicht kompatibel zu Objekten des Typs *stackA.T* oder *stack.T* sind. Auch Objekte des Typs *stackA.T* und *stack.T* sind nicht untereinander austauschbar, da der Typüberprüfer die Kompatibilität der Implementationen nicht zusichern kann.

5.2.4 Rautenimport

Das bisher vorgestellte Modell der Strukturierung mittels Modulen und Schnittstellen bietet dem Programmierer große Flexibilität insbesondere in bezug auf die Kombination von Modulen zu ablauffähigen Programmen.

Abbildung 5.12 stellt zwei Programmkonfigurationen gegenüber, die die besondere Problematik der abstrakten Datentypen aufdecken. Der vollständige Programmcode des Beispiels ist in Anhang D wiedergegeben. In Teil a wird das Modul *stack* über mehrere Pfade importiert. Werden die Module *stack* bzw. *fastStack* über die Module *kreditor* und *debitor* beide unter dem Namen *stack* sichtbar, so muß der Typüberprüfer feststellen, daß es sich hierbei im Fall a um dasselbe Modul handelt und im Fall b nicht. Er sollte die beiden in Abbildung 5.12 dargestellten Fälle a und b unterscheiden können.



a) Rautenimport b) Import unterschiedlicher Module unter gleichem Namen

Abbildung 5.12: Rautenimport.

Die nachfolgende Sequenz zeigt, wie der Fall a behandelt werden kann. Hierbei ist zu berücksichtigen, daß, wie auch der Übersetzungvorgang selbst, der Bindevorgang durch Werkzeuge unterstützt werden sollte, um einerseits die Reihenfolge der Anweisungen einzuhalten und andererseits die fehleranfällige Formulierung der Aufrufe zu automatisieren.

```
Let Stack = Stack(:StdType :StdIde stdIde);
let stack :Stack = stackLINK(:StdType stdType
                             :StdIde stdIde
                             :List list :Stack);
Let Kreditor = Kreditor(:StdType stdType
                             :StdIde stdIde
                             :List list
                             :Stack stack);
Let Debitor = Debitor(:StdType stdType
                           :StdIde stdIde
                           :List list
                           :Stack stack);
let kreditor :Kreditor = kreditorLINK(:StdType stdType
                                      :StdIde stdIde
                                      :List list
                                      :Stack stack
                                      :Kreditor);
let debitor :Debitor = debitorLINK(:StdType stdType
                                   :StdIde stdIde
                                   :List list
                                   :Stack  stack
                                   :Debitor);
Let Mandant = Mandant(:StdType stdType
                      :StdIde stdIde
                      :List list
                      :Stack stack
                      :Debitor debitor
                      :Kreditor kreditor);
let mandant :Mandant = mandantLINK(:StdType stdType
                                   :StdIde stdIde
                                   :List list
                                   :Stack stack
                                   :Debitor debitor
                                   :Kreditor kreditor
                                   :Mandant);
```

In der semantischen Analyse kann der Typüberprüfer durch Vergleich der Objektidentitäten herausfinden, daß der Typ *stack.T* aus *debitor* mit dem gleichnamigen Typ aus *kreditor* identisch ist. Dies ist anhand der Parameter obiger Ausführungssequenz leicht nachzuvollziehen. Im Modul *mandant* ist eine Anwendung der Form

```
debitor.put(kreditor.get())
```

typkorrekt. Nachfolgend sind Anweisungen wiedergegeben, die dem in Abbildung 5.12 gezeigten Fall b entsprechen:

```
Let Stack = Stack(:StdType :StdIde stdIde);
let stack :Stack = stackLINK(:StdType stdType
                          :StdIde stdIde
                          :List list :Stack);
let fastStack :Stack = stackLINK(:StdType stdType
                              :StdIde stdIde
                              :List list :Stack);
Let Kreditor = Kreditor(:StdType stdType
                             :StdIde stdIde
                             :List list
                             :Stack stack);
Let Debitor = Debitor(:StdType stdType
                           :StdIde stdIde
                           :List list
                           :Stack fastStack);
let kreditor :Kreditor = kreditorLINK(:StdType stdType
                                      :StdIde stdIde
                                      :List list
                                      :Stack stack
                                      :Kreditor);
let debitor :Debitor = debitorLINK(:StdType stdType
                                   :StdIde stdIde
                                   :List list
                                   :Stack  fastStack
                                   :Debitor);
Let Mandant = Mandant(:StdType stdType :StdIde stdIde
                      :List list
                      :Stack stack
                      fastStack
                      :Debitor debitor
                      :Kreditor kreditor);
let mandant :Mandant = mandantLINK(:StdType stdType
                                   :StdIde stdIde
                                   :List list
                                   :Stack stack
                                   :Stack fastStack
                                   :Debitor debitor
                                   :Kreditor kreditor
                                   :Mandant);
```

Schon durch die veränderte Parameterliste von *mandant* und *Mandant* wird der Unterschied zu

Fall a deutlich: Schnittstelle *Mandant* und Modul *mandant* sind transitiv sowohl von *stack* als auch von *fastStack* abhängig. Der Typoperator *Mandant* muß zusätzlich mit *fastStack* parametrisiert werden, und die Funktion *mandantLINK* erfordert ebenfalls *fastStack* als weiteres Argument. Zwar ändert sich am Schnittstellen- und Modulrumpf nichts, der Typüberprüfer weist die über *kreditor* und *debitor* referenzierbaren *stack.T* jedoch als inkompatibel zurück, da es sich einmal um *stack.T*, im anderen Fall um *fastStack.T* handelt.

Aufbauend auf den Basisprimitiven der Sprache TLMIN zur Manipulation von Sichtbarkeitsbereichen zeigt dieser Abschnitt, wie daraus das semantisch reichhaltigere Konzept der Modulen und Schnittstellen aufgebaut werden kann. Zwar kommt diese Abbildung nicht gänzlich ohne Werkzeugunterstützung aus, es sind jedoch zur Realisierung keine Eingriffe in das TLMIN Kernsystem notwendig. Die unterschiedlichen Bindemechanismen zeigen, welche Flexibilität dem Programmierer in der Wahl seiner Programmkonfiguration, der Zusammenstellung von übersetzten Einheiten, zur Verfügung steht. Auch hier ist allerdings eine Werkzeugunterstützung zum effizienten Arbeiten unerläßlich.

5.3 Umgebungen

Umgebungen (*environments*) sind ein weiteres Strukturierungsmittel. In Abgrenzung zu den Basisprimitiven aus Abschnitt 5.1 sind Umgebungen auf einer höheren Ebene angesiedelt, beeinflussen jedoch ebenfalls die Sichtbarkeit von Bezeichnern. Sie gruppieren Softwareeinheiten bzw. Namen, wie sie in Abschnitt 5.2 modelliert werden, zu größeren Komplexen oder Systemen. Diese Gruppen können dazu benutzt werden, verschiedenen Entwicklern unterschiedliche Mengen von Softwareeinheiten zur Verfügung zu stellen und damit die parallele Softwareentwicklung zu unterstützen. Sie können weiterhin dazu verwendet werden, Gruppen nach unterschiedlichen semantischen Gesichtspunkten zu formen und so die Komplexität eines großen Softwaresystems zu beherrschen.

Eine Umgebung faßt für jeden in ihr abgelegten Wert die unterschiedlichen beschreibenden Informationen und (Zwischen-) Repräsentationen des Compilers zusammen. Ihre Datenstruktur zusammen mit entsprechenden Operationen sind über die in Anhang C gezeigte Schnittstelle erreichbar. Sie sind nicht direkt Teil der Sprache TLMIN, können aber durch eine Syntaxerweiterung mittels Schlüsselwörtern angesprochen werden und dem Programmierer den Eindruck vermitteln, daß sie zur Sprache selbst gehören. Umgebungen sind in TLMIN eine Möglichkeit, Softwareeinheiten wie z.B. Module in größere Einheiten zusammenzufassen und damit zu strukturieren. Für den Compiler legen sie während des Übersetzens die Menge an sichtbaren Bezeichnern fest. Sie übernehmen damit ebenfalls die Funktion der Toplevel in TL Systemen [Matthes 1993], verfolgen jedoch einen allgemeineren Ansatz, da sie die folgenden Eigenschaften aufweisen:

- Umgebungen können sowohl unveränderliche als auch variable Bindungen enthalten.
- Die Typinformation ist mit jedem Eintrag persistent abgelegt. Dadurch entfällt die Notwendigkeit, explizite Mechanismen für dynamische Typen der in Umgebungen enthaltenen

Bezeichner anzubieten.

- Umgebungen können beliebig strukturiert werden. Insbesondere sind Schachtelungen denkbar, die zu Strukturen ähnlich denen eines Dateisystems führen. Dies folgt daraus, daß Bindungen zwischen den Umgebungen aufgebaut werden können.

- Komponenten können zwischen Umgebungen beliebig verschoben oder kopiert werden. Namensänderungen von Komponenten sind möglich und verletzen wie die vorgenannten Operationen Verschieben und Kopieren die Typsicherheit nicht.

5.3.1 Aufbau

Umgebungen werden im Übersetzer von der Phase der Typüberprüfung bis zur Ausführung des virtuellen Maschinencodes in der Tycoon Maschine benutzt. Sie stellen ergänzend zu den verschiedenen Zwischen- und Coderepräsentationen die Verbindung zwischen dem Compiler als statischer Komponente einerseits und den aktuellen dynamischen Daten (Programme, Ausdrücke) andererseits her, indem sie Informationen, wie z.B. den Typ von referenzierten Bezeichnern, enthalten, auf die sich der Übersetzer während des Transformationsvorganges beziehen kann. So wird im nachfolgenden Beispiel im zweiten Ausdruck Bezug auf einen zuvor gebundenen Bezeichner *a* genommen:

```
let a = 42
⇒ 42 :Int

let b = a * 2
⇒ 84 :Int
```

Der Compiler muß bei der semantischen Analyse des zweiten Ausdrucks Typinformationen über verwendete Bezeichner, in diesem Fall das *a* und die Zahl *2* bekommen, um die Korrektheit des Ausdrucks zusichern zu können. Bei der Zwischencode-Generierung benötigt der Übersetzer Informationen über die Art, wie auf den Speicherplatz, der durch *a* repräsentiert wird, zugegriffen wird. So unterscheidet sich z.B. der Zugriff auf eine Variable von der Anwendung einer Funktion. Aus dieser Kenntnis wird der entsprechende TML-Code generiert, der den Zugriff ausführt. Der TML-Code dient der nachfolgenden Phase zur Erzeugung des von der virtuellen Maschine ausführbaren Bytecodes. Umgebungen dienen dazu, jeder Phase die Information über bereits typgeprüfte, zwischencodegenerierte oder ausgeführte Ausdrücke bereitzustellen, die sie zur Erledigung ihrer Ausgabe benötigt.

Eine Umgebung setzt sich aus drei Teilkomponenten zusammen.

Typkomponente: Sie umfaßt die Signaturen aller sichtbaren Bezeichner wie Funktionen, Variablen etc. Unter Signaturen wird in diesem Zusammenhang die Repräsentation dieser Signaturen im Syntaxbaum verstanden. Konkret sind in der Typkomponente somit Verweise auf Teilsyntaxbäume enthalten.

Wertkomponente: Die Codegenerierung übersetzt die Repräsentation des Syntaxbaums in die Darstellung als TML-Baum. Die Knoten dieses Baums werden in der Wertkomponente abgelegt.

Laufzeitkomponente: Nach Erzeugung und Ausführung des TVM-Codes werden die evaluierten Werte in die Laufzeitkomponente eingetragen und stehen damit nachfolgenden Ausführungsschritten zur Verfügung.

Umgebungen erweitern damit das Konzept der Symboltabelle, wie Abbildung 5.13 in Anlehnung an Darstellung 4.3 auf Seite 51 und an entsprechende Darstellungen in [Aho et al. 1988] verdeutlicht. Zum einen enthalten sie Informationen von bereits übersetzten Ausdrücken und Programmen und speichern diese dauerhaft. Zum anderen werden Umgebungen im TLMIN System auch zur Laufzeit benötigt, da sie in der Laufzeitkomponente Verweise auf evaluierte Ausdrücke enthalten und somit als Schnittstellen zu diesen fungieren. Als zusätzliche Erweiterung gegenüber normalen Symboltabellen kann das Konzept angesehen werden, daß der Compiler mit Umgebungen parametrisiert werden kann. Das bedeutet, daß ein Ausdruck vom Compiler stets gegen eine bestimmte Umgebung übersetzt werden muß und auch in Abhängigkeit von der Umgebung verschiedene Ergebnisse errechnen kann.

Konzeptuell faßt eine Umgebung für jeden Wert die entsprechende Information des Compilers zusammen. Umgebungen nehmen Bezeichner zusammen mit ihrer Typinformation, ihrer Wertinformation[4] und der Laufzeitrepräsentation auf. Sie werden vom Compiler sequentiell mit den errechneten Werten gefüllt. Bezüglich des Übersetzungsvorganges ergeben sich daraus einige Integritätsbedingungen, die auf Umgebungen definiert sind.

Der Übersetzungsvorgang wird durch die Compilerschnittstelle in mehrere Funktionen aufgeteilt, deren Ergebnis in der entsprechenden Zwischenrepräsentation (s. Abbildungen 5.13 und 4.3 auf Seite 51) jeweils als Argument des nächsten Schrittes auftritt. Die Übersetzung erfolgt dabei jeweils gegen eine Umgebung, die ebenfalls als Parameter der Funktionen auftritt. Im Beispiel von Abbildung 5.14 ist dies die Umgebung *Aenv* für den linken und *Benv* für den rechten Ausführungsstrang. Wie in Abbildung 5.14 ebenfalls zu sehen, können sich die Übersetzungsschritte unterschiedlicher Ausdrücke durch bestimmte Ausführungsreihenfolgen überlappen. Die zur Übersetzung eines Ausdruckes notwendigen Funktionsaufrufe werden nicht notwendigerweise sequentiell ausgeführt, sondern können mit den Funktionsaufrufen eines oder mehrerer anderer Übersetzungsvorgänge verzahnt sein[5]. So wird nach dem Parsen des Ausdrucks *A* (*parse(A)*) zunächst der Ausdruck *B* geparst, semantisch überprüft, in Zwischencode übersetzt und virtueller Maschinencode generiert, bevor diese Schritte inklusive der Ausführung auch für den

[4]In dieser Arbeit wird der Begriff Wertkomponente benutzt, um die Massendatenstruktur zu beschreiben, die die vom Codegenerator erzeugten Knoten enthält. Er ist abgeleitet vom Variablennamen *valueEnv*, der in der Implementierung des TLMIN Compilers Verwendung findet. Es sind auch andere Bezeichnungen, wie z.B. Variablenkomponente, denkbar. Die Belegung der in dieser Datenstruktur gespeicherten Objekte mit dem Begriff Wert scheint jedoch aus dem Grund angebracht, da es sich um Zugriffsbeschreibungen für Werte (Literale, Funktionen usw.) handelt.

[5]Die korrekte Reihenfolge der Funktionsaufrufe eines Übersetzungsvorganges ist dadurch gewährleistet, daß als Parameter jeweils das Funktionsergebnis des vorherigen Schrittes übergeben werden muß. Die darüber hinaus korrekte Benutzung der Übersetzerfunktionen und die Bereitstellung der korrekten Argumente liegt in der Verantwortung des Programmierers.

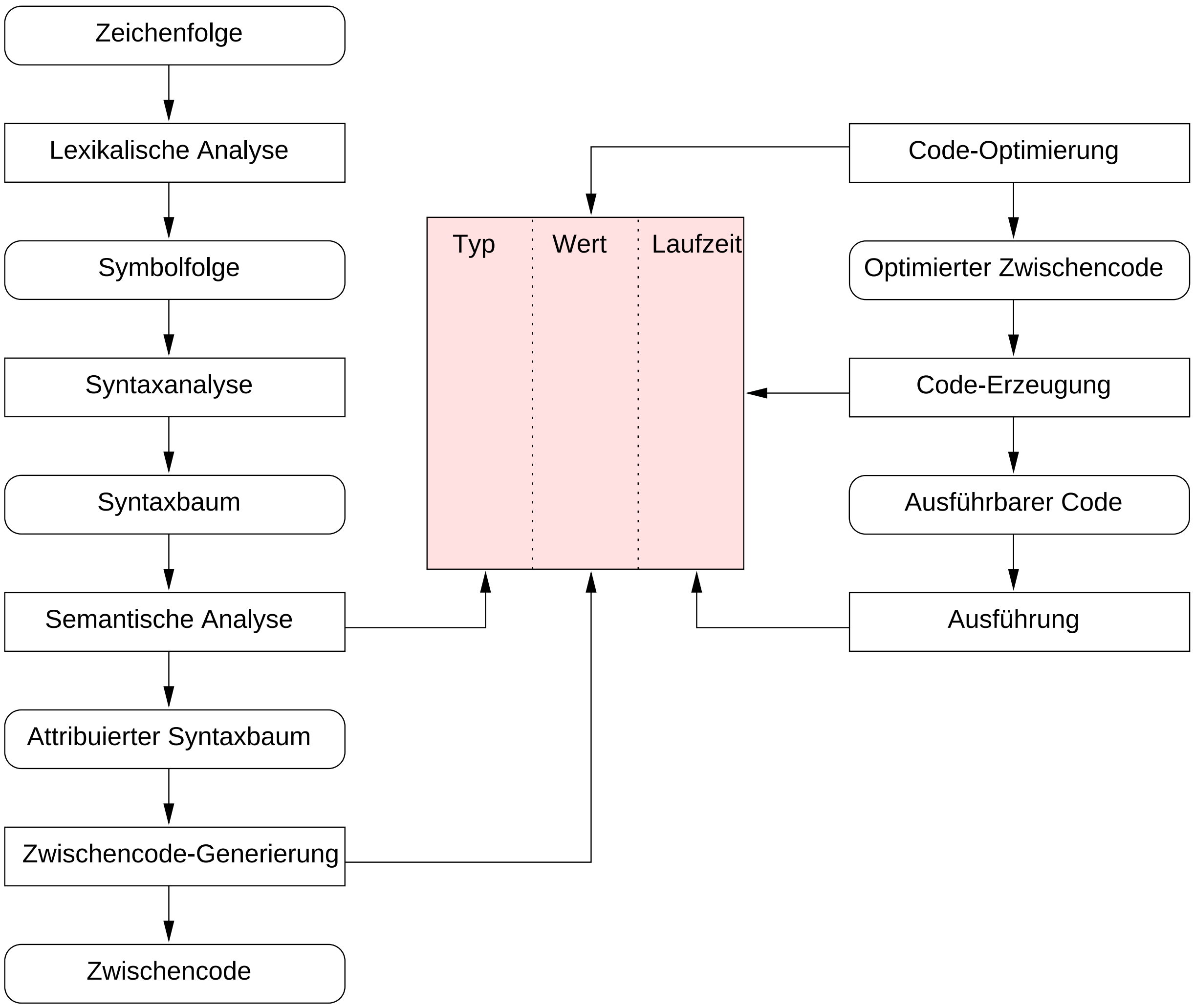


Abbildung 5.13: Compilerphasen mit Zwischenrepräsentationen und Umgebung.

Ausdruck *A* durchgeführt werden. Dennoch können sich diese Vorgänge auf nur eine Umgebung beziehen. Eine Umgebung muß sich in einem solchen Fall nach jedem Übersetzungsschritt in einem konsistenten Zustand befinden; d.h., obwohl die Umgebung sequentiell gefüllt wird, müssen die Einzelkomponenten eines Wertes jeweils zuordenbar sein.

5.3.2 Anwendungen

Einsatzmöglichkeiten von Umgebungen ergeben sich in erster Linie aus den beschriebenen Anwendungen als über den Basisprimitiven angesiedeltes Strukturierungsmittel und durch die erweiterte Sicht als Symboltabelle als Compilerhilfsmittel. Auch in Zusammenhang mit der Verwirklichung von (grafisch unterstützten) Entwicklungsumgebungen und der Realisierung einer teamorientierten Entwicklungsunterstützung ergeben sich für Umgebungen insbesondere in Verbindung mit der Compilerschnittstelle (s. Abschnitt 4.3) weitere Anwendungsmöglichkeiten.

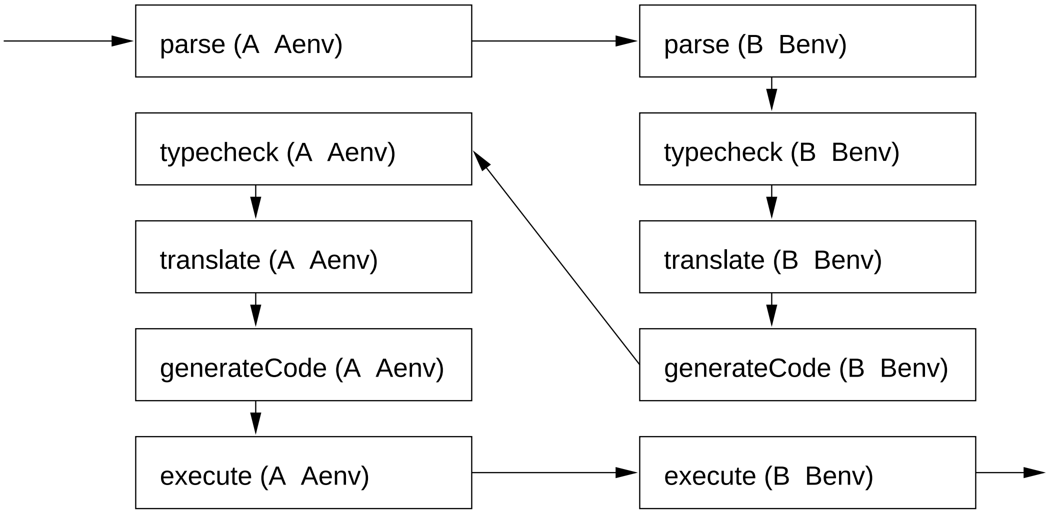


Abbildung 5.14: Überlappende Aufrufe der Compilerschnittstelle.

Die grundlegenden Funktionen auf Umgebungen können der Schnittstelle in Anhang C entnommen werden. Grundsätzlich können neue Einträge auf zwei unterschiedliche Arten in eine Umgebung gelangen:

- Durch Übersetzen eines Ausdruckes trägt der Compiler die von ihm generierten Objekte in die als Parameter übergebene Umgebung ein.
- Bereits vorhandene Bezeichner werden von einer Umgebung in eine weitere kopiert oder verschoben.

Mit Umgebungen lassen sich hierarchische oder andere azyklische Strukturen aufbauen. Die in Abbildung 5.15 dargestellte Situation wird durch die folgenden Anweisungen erreicht.

```
let root = environment.new()
```

Die weiteren Anweisungen werden in der zuvor erzeugten Umgebung *root* übersetzt. Dazu wird die Existenz einer Funktion *compileString* angenommen, die die in Abschnitt 4.3 beschriebenen Funktionen der Compilerschnittstelle mit der als Argument übergebenen Umgebung aufruft und als Ergebnis die um die entsprechende Bindung erweiterte Umgebung zurückliefert.

```
let root = compileString("let standard = environment.new()"
                          root)
let root = compileString("let bulk = environment.new()"
                          root)
let root = compileString("let graphic = environment.new()"
                          root)
```

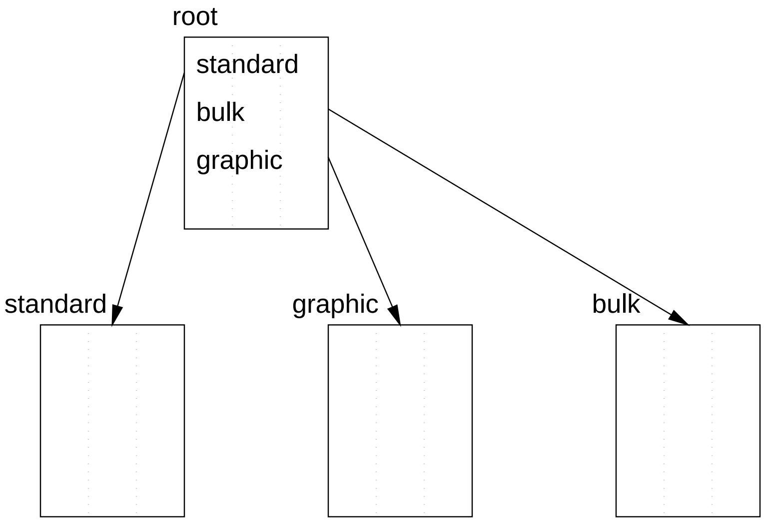


Abbildung 5.15: Baumartige Struktur von Umgebungen.

Weitere Freiheitsgrade erhält der Programmierer dadurch, daß Komponenten einer Umgebung umstrukturiert werden können. So sind die Umgebungen *standard*, *graphic* und *bulk* in der Umgebung *root* enthalten. Sie treten dort als Bindungen auf, deren Wert vom Typ *environment.T* (s. Anhang C) ist.

Im TLMIN System können durch Erzeugen entsprechender Bindungen mehrere parallele Umgebungen gehalten werden (s. Abbildung 5.17). Für solche Situationen ist es wünschenswert, einzelne Komponenten aus einer Umgebung herauslösen und in eine weitere wiedereingliedern zu können. Denkbar ist ein Szenario, in dem ein Systemprogrammierer in der Umgebung *root* entwickelt und einem Anwendungsentwickler fertige Komponenten in der Umgebung *user* zur Verfügung stellt. Folgende Operation kopiert den Bezeichner *set* aus der Umgebung *bulk* in die Umgebung *user*:

```
let user = environment.copyBinding(bulk user "set")
```

In der Umgebung *user* ist nun tatsächlich nur der Bezeichner *set* sichtbar, obwohl zu dessen Erzeugung, insbesondere dessen semantischer Überprüfung, auch andere Bezeichner notwendig waren. Dies ist vor dem Benutzer der Umgebung *user* verborgen. Er kann jedoch Code entwickeln, der auf *set* aufbaut.

Durch eine gleichzeitige oder spätere Umbenennung des Bezeichners *set* erhält der Programmierer weitere Kontrolle an die Hand. Wird z.B. in der Umgebung *user* eine andere Namenskonvention verfolgt, als das in der Umgebung *bulk* der Fall ist, so kann *set* auf diese Weise angepaßt werden:

```
let user = environment.copyBindingAs(bulk user
                "set" "Massendaten_Menge")
```

Da Objektreferenzen zwischen den Komponenten von den Manipulationen innerhalb der Umgebungen unberührt bleiben, ist es nicht notwendig, daß der Programmierer alle zur Übersetzung notwendigen Komponenten kopiert. Nur die Komponenten, die der Benutzer tatsächlich sehen soll, müssen kopiert werden. Auf der Ebene der Objektreferenzen ergibt sich damit die in Abbildung 5.16 dargestellte Situation, die am Beispiel der Komponente *set* und der beiden Umgebungen *bulk* und *user* verdeutlicht, daß innerhalb der Umgebungen nur Verweise auf die eigentlichen Objekte abgelegt sind.

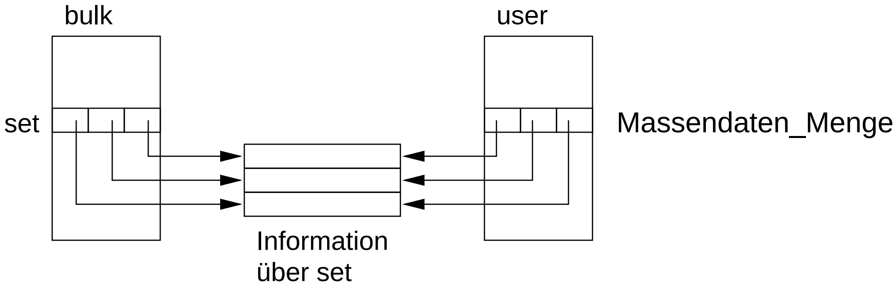


Abbildung 5.16: Zusammenhang zwischen Umgebungen und Objektreferenzen.

In Zusammenhang mit den in Abschnitt 5.2 und folgende vorgestellten Syntaxerweiterungen für Module und Schnittstellen und deren werkzeugmäßige Unterstützung ergibt sich an dieser Stelle eine Schwierigkeit. Zur Umsetzung der dort vorgestellten Syntax für Schnittstellen und Module müssen alle transitiv erreichbaren Bezeichner gefunden und als Parameter verfügbar sein. Dies ist, wenn sich nur das benutzte Modul in einer Umgebung befindet, nicht der Fall. Dieser Vorgang könnte von einem Werkzeug durchgeführt werden, welches die erforderlichen Bezeichner ggf. nach bestimmten Regeln in anderen Umgebungen aufsucht. Für den Übersetzer stehen diese jedoch noch nicht zur Verfügung, da er immer nur mit einer Umgebung parametrisiert wird, die Namen und die zugehörigen Informationen und Repräsentationen aber in unterschiedlichen Umgebungen vorliegen. Die Lösung kann darin bestehen, die Übersetzung in einer weiteren (temporären) Umgebung durchzuführen, die alle notwendigen Bezeichner enthält. Nach Abschluß des Übersetzungsvorganges wird der gewünschte Bezeichner in die entsprechende Zielumgebung kopiert. Die temporäre Umgebung kann für weitere Übersetzungsvorgänge erhalten bleiben.

Weiterhin ist beachtenswert, daß die Übersetzung von Ausdrücken, die innerhalb ihrer Originalumgebung[6] geändert werden, durch verschobene Bezeichner nicht reproduzierbar werden kann. Durch die folgende Anweisung und eine angenommene Abhängigkeit des Moduls *bag* von dem Modul *set* kann ersteres nach einer Änderung nicht erneut übersetzt werden.

```
let user = environment.moveBinding(bulk user "set")
```

Dennoch bleibt das nicht geänderte Modul *bag* weiterhin funktionsfähig, was auf die Verwendung von Objektreferenzen zurückzuführen ist.

[6]Hiermit ist die Umgebung gemeint, in der der entsprechende Ausdruck übersetzt wurde.

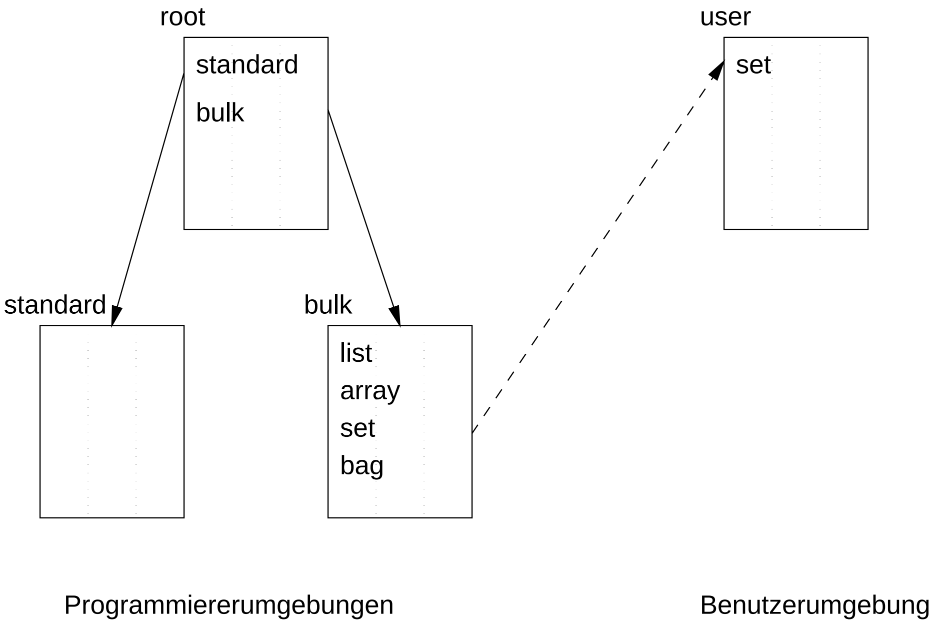


Abbildung 5.17: Operationen zwischen parallelen Umgebungen.

Die Ausführungen zu Umgebungen beziehen sich auf ein Szenario, in dem sich alle Operationen in einem Objektspeicher abspielen, d.h. Bezeichner, die von einer Umgebung in eine andere kopiert bzw. verschoben werden, verbleiben im selben Objektspeicher. Zusätzliche Dynamik erhält dieses Szenario jedoch durch die Verwendung von unterschiedlichen Instanzen eines Tycoon Systems und somit Objektspeichern und in Zusammenhang mit Netzwerkreferenzen [Ramme 1997], was in dieser Arbeit jedoch nicht betrachtet wird. Insbesondere die Entwicklung von Software durch mehrere Programmierer in einem Team kann hierdurch profitieren.

5.4 Anwendungsszenario

In diesem Abschnitt werden die bislang getrennt vorgestellten Konzepte und Mechanismen in einen größeren Zusammenhang gerückt. Dabei soll das Hauptaugenmerk auf das Zusammenspiel von Werkzeugen, wie die Modulverwaltung auf der obersten Ebene, Compilerschnittstelle und Umgebungen auf einer darunterliegenden Ebene und Basisprimitiven zur Systemstrukturierung und daraus kombinierte höhere Konzepte auf der untersten Ebene gerichtet werden.

Das Ziel dieser Arbeit ist es, die in Kapitel 1 formulierten Prinzipien zu unterstützen. Unterstützung zur Erreichung dieses Ziels wird einerseits durch Strukturierungsprimitive (Abschnitt 5.1.2 bis 5.1.5) in der Programmiersprache und andererseits durch Werkzeuge (Abschnitt 4.4) bereitgestellt. Die Strukturierungsprimitive sind so gewählt, daß sich durch Kombination neue, semantisch reichhaltigere Strukturierungsmechanismen aufbauen lassen. In dieser Arbeit sind dies Module und Schnittstellen (Abschnitt 5.2.1 und 5.2.2).

Ein Entwickler kann sein Programm in Module und Schnittstellen zerlegen und diese unabhängig voneinander entwickeln und implementieren, da sie separat übersetzbar sind. Er wird damit direkt bei der Erreichung der Prinzipien Abstraktion, Strukturierung, Modularisierung und

Wiederverwendbarkeit unterstützt. Durch die Formulierung der Module und Schnittstellen wird z.B. für die importierten Bezeichner das Prinzip Lokalität erreicht. Hierarchisierung, in diesem Fall speziell in der Form eines azyklischen, gerichteten Graphen, wird durch die nicht wechselseitig zugelassenen Importbeziehungen erreicht. Unterstützung bei der Formulierung der Module und Schnittstellen, insbesondere bei der Ermittlung der transitiven Importe, erhält der Programmierer durch die Modulverwaltung auf Werkzeugebene.

Zur Verwaltung der Module und Schnittstellen und zum Übersetzen, Binden und Ausführen stehen der Modulverwaltung die Compilerschnittstelle (Abschnitt 4.3) und die Schnittstelle zu Umgebungen (Abschnitt 5.3) zur Verfügung. Sie stellen das Bindeglied zwischen der Modulverwaltung, die als Anwendungsprogramm abläuft, und dem TLMIN System dar, welches darüber bestimmte Dienste anbietet.

Die Modulverwaltung bietet dem Programmierer über entsprechende Funktionen den Dienst an, für Module und Schnittstellen die (transitiven) Importe zu ermitteln und sie anschließend in Abhängigkeit dieser Importe zu übersetzen und zu binden. Beim Binden stehen verschiedene Optionen für die Konfiguration von Modulen und Schnittstellen zur Verfügung (Abschnitt 5.2.3), die es dem Entwickler z.B. erlauben, Module nicht nur mit Modulen zusammenzubinden, die den gleichen Namen, wie die in der Importliste erwähnten, besitzen. Vielmehr läßt sich ein Modul mit beliebigen Modulen und Schnittstellen zusammenbinden, die dem entsprechenden Typ bzw. einem Subtyp genügen, wobei Einschränkungen beim Datenaustausch zwischen abstrakten Datentypen hingenommen werden müssen (Abschnitt 5.2.4).

Die Modulverwaltung ist weiterhin in der Lage, die Module und Schnittstellen in größeren Einheiten, Umgebungen, zu verwalten (Abschnitt 5.3). Diese kann ein einzelner Programmierer zur Strukturierung seiner Softwareeinheiten einsetzen. Mehrere Programmierer eines Teams können über Umgebungen ihre Module und Schnittstellen austauschen.

5.5 Zusammenfassung

In diesem Kapitel werden Strukturierungsmechanismen vorgestellt, die einerseits in der Sprache TLMIN eingebaut sind, andererseits als Dienst über eine Schnittstelle angeboten werden. Sie bieten im Vergleich zu den in Kapitel 3 vorgestellten und bewerteten Mechanismen, die fest in Programmiersprachen integriert sind, ein höheres Maß an Flexibilität in bezug auf ihre Kombinierbarkeit und damit auch auf ihre Einsatzmöglichkeiten.

Die zur Verfügung stehenden orthogonal kombinierbaren Operationen zur Manipulation von Namensräumen bzw. Sichtbarkeitsbereichen erlauben es, eine große Zahl von weiteren höheren Konzepten aufzubauen. Der in TLMIN gewählte Ansatz unterscheidet sich hierdurch grundlegend von denen in Kapitel 3, da keine Konzepte wie Module oder Klassen fest in die Sprache integriert sind, sondern aus den Basisprimitiven aufgebaut werden. Das läßt die Option offen, das TLMIN System durch eine Syntaxerweiterung dergestalt um Schlüsselwörter zu ergänzen, daß die Strukturierungsmechanismen einfach, effizient, d.h. unter Vermeidung von Fehlern und Schreibaufwand benutzt werden. Für den vorgestellten Fall der Module und Schnittstellen ist, wie in Abschnitt 5.4 gezeigt, weiterhin eine Werkzeugunterstützung zum praktischen Einsatz empfehlenswert, da die transitiven Importe nicht allein aus der Syntax hervorgehen. Folglich ist ein TLMIN Basissystem ohne Erweiterung und ergänzende Werkzeuge nur bedingt als Programmiersprache für große Systeme geeignet, sondern sollte, wie in Abschnitt 5.2 gezeigt, durch Abbildungen von Syntaxerweiterungen auf die Basissprache angepaßt werden. Dies kann sehr flexibel und ohne Verlust der Kompatibilität des erzeugten Codes und der erzeugten Zwischenrepräsentationen des Compilers geschehen.

In bezug auf Werkzeugunterstützung und syntaktische Erweiterungen gilt Gesagtes auch für Umgebungen. Sie müssen durch zusätzliche, dem Anwendungsfall angepaßte, syntaktische Konstrukte verfügbar gemacht werden, und ihre Verwendung sollte mit Hilfe von Werkzeugen erfolgen. Umgebungen erfüllen jedoch noch eine weitere Aufgabe, indem sie über bereits bestehende Softwareeinheiten eine Verwaltung anbieten, wie sie auch in [Booch 1994] gefordert wird. Umgebungen unterstützen damit zwei Aspekte der Entwicklung großer Softwaresysteme:

- Programmieren in Teams mehrerer Entwickler durch Zuteilung unterschiedlicher Umgebungen mit der Möglichkeit des gegenseitigen Austauschs von Softwareeinheiten.

- Parametrisieren des Übersetzungsvorganges, indem dem Compiler durch eine Umgebung eine Menge an Namen inklusive der damit verbundenen beschreibenden Information und Repräsentationen übergeben wird, gegen die er die Übersetzung durchführt.

Die Flexibilität der vorgestellten Ansätze zeigt, daß diese durch Werkzeuge unterstützt werden müssen, damit sie effizient und sicher genutzt werden können. Als Beispiel sei auf die in Abschnitt 5.2.4 beschriebene Möglichkeiten zur Programmkonfiguration verwiesen, die es erlauben, Module ohne Neuübersetzung umzukonfigurieren. Hierzu wird in Abschnitt 4.4 ein Ansatz vorgestellt, der die allgemeine Anbindung eines solchen Werkzeugs darstellt. Abschließend kann festgehalten werden, daß durch die Aufnahme von Operationen zur Manipulation von Sichtbarkeitsbereichen in die Sprache TLMIN und durch die auf höherer Ebene angesiedelten Umgebungen dem Programmierer Möglichkeiten zur Systemstrukturierung an die Hand gegeben werden.

Kapitel 6

Zusammenfassung

In dieser Arbeit werden Mittel zur Systemstrukturierung, speziell zur Manipulation von Sichtbarkeitsbereichen, und zur Anbindung von Werkzeugen an ein persistentes Objektsystem unter Verwendung einer Compilerschnittstelle untersucht. Die enge Verzahnung dieser beiden Bereiche macht die Notwendigkeit einer Entwicklungsumgebung zur effizienten Systemstrukturierung und -konfiguration deutlich. Anhand der konkreten Implementierung des TLMIN Systems wird die Aufgabenteilung zwischen Entwicklungswerkzeug als Anwendungsprogramm einerseits und Sprachkonstrukten als Bestandteil des Compilers andererseits gezeigt.

Auf Sprachebene wird der Fokus auf die Behandlung von Sichtbarkeitsbereichen und den damit in Zusammenhang stehenden Namensräumen gelegt. Die identifizierten Basisoperationen zur Manipulation von Sichtbarkeitsbereichen sind in TLMIN implementiert. Sie bilden die Grundlage für höhere Strukturierungskonzepte wie Module und Schnittstellen. Umgebungen sind als reflektive Schnittstelle des TLMIN Systems realisiert. Beide Konzepte zusammen bilden die Grundlage für die beispielhaft implementierte Modulverwaltung, welche die mit Hilfe von Syntaxerweiterungen und Basisprimitiven formulierten Module und Schnittstellen in Umgebungen verwaltet. Sie bedient sich weiterhin der ebenfalls reflektiv bereitgestellten Compilerschnittstelle, um die Module und Schnittstellen zu übersetzen und zu binden.

Beim Entwurf der Compilerschnittstelle wird darauf geachtet, daß sie die einzelnen Phasen des Übersetzungsvorganges abbildet. Auf diese Weise können die in einem Compiler üblicherweise vorhandenen Zwischenrepräsentationen, die als Funktionsergebnis und -argumente auftreten, verknüpft, ausgewertet und persistent abgelegt werden. Dazu zählen z.B. der Syntaxbaum oder der ausführbare Code. Mit entsprechenden Werkzeugen kann auf diesen Datenstrukturen gearbeitet werden.

Das Bindeglied zwischen Sprachkonstrukten und Compilerschnittstelle ist in den Umgebungen zu sehen. Sie bieten auf der einen Seite die Möglichkeit, Bezeichner bzw. Bindungen zusammenzufassen und auf beliebige Weise zu strukturieren. Sinnvoll erscheinen hier besonders baumartige Strukturen, die ähnlich einem Dateisystem Namen organisieren. Auf der anderen Seite fassen Umgebungen die Menge an Bezeichnern zusammen, die der Compiler für die Übersetzung eines Ausdrucks benötigt. An dieser Stelle ist allerdings die zu fragen, ob das gegenwärtig verwendete Modell, daß der Compiler die referenzierten Bezeichner nur in einer flachen Struktur wie einer einzelnen Umgebung aufsuchen kann, ausreichend ist. Bezeichner, die sich in mehreren Umgebungshierarchien befinden, können in diesem Modell in ihrer Gesamtheit bisher nicht dem Übersetzungsvorgang bereitgestellt werden.

6.1 Kommerzielle Relevanz

Ein Vorteil der vorgestellten Modelle und Konzepte liegt in der Tatsache, daß mit Hilfe der flexiblen Möglichkeiten in bezug auf unterschiedliche Strukturierungskonzepte eine sehr effiziente Anpassung der Programmiersprache an die in der Entwurfsphase angewendete Methode vorgenommen werden kann. In diesem Zusammenhang ist unter effizient in erster Linie zu verstehen, daß die Verluste an Struktur, Verständnis, Wartbarkeit und Änderungsfreundlichkeit gering gehalten und damit die bereits in der Einleitung aufgeführten Prinzipien Abstraktion, Strukturierung, Hierarchisierung, Modularisierung, Lokalität und Wiederverwendbarkeit in hohem Maße unterstützt werden. Durch die Verwendung von Basisprimitiven, die unter Zuhilfenahme von Syntaxerweiterungen und durch Kombination zu neuen Konstrukten geformt werden können, bleibt die Kompatibilität zu vorhandenem Code gewährleistet. Dies ist unter Kostengesichtspunkten ein großer Vorteil, da auch bei einem Wechsel der Spezifikations- oder Entwurfsmethodik bereits geschriebener Code nicht nutzlos ist. Vielmehr ist die Integration alter Programme durch die Kompatibilität der weiterhin benutzten Kernsprache gegeben. Insbesondere die Integration, Pflege und Überarbeitung bestehenden Codes stellt in der informationstechnologischen Praxis einen Kostenfaktor dar. Hier gibt diese Arbeit Ansätze zur Minderung dieser Kostenbelastung.

Ein weiteres Ergebnis dieser Arbeit sind die Erkenntnisse aus der Untersuchung der Strukturierungskonzepte von C++ und der Entwicklungsumgebung des SAP R/3 Systems bzw. der Sprache ABAP/4, bei der durch die hohe Integration dieses Systems bedingten Verzahnung von Sprache und Umgebung auch die Strukturierungskonzepte betrachtet werden. Im Fall von C++ kann bemerkt werden, daß besonders durch die erzwungene Kompatibilität zum nicht objektorientierten Vorgänger C eine Reihe semantisch nicht sauber definierter Mechanismen in die Sprache Einzug gehalten haben bzw. nicht beseitigt wurden. So beseitigt z.B. die Verwendung von Namensräumen auf Sprachebene ein Strukturierungsproblem großer Systeme nur unzureichend, da es sich lediglich um ein Schema zur Benennung handelt, welches das System direkt unterstützt, jedoch keine Möglichkeit bietet, Bezeichner zu verstecken. Gleichzeitig wird deutlich, daß der Zwang zur Kompatibilität zu C viele dieser Mängel verursacht. Daraus läßt sich abermals die Notwendigkeit und die Richtigkeit des in dieser Arbeit gewählten Ansatzes erkennen, da hier trotz Verwendung neuer Konzepte alter Code kompatibel bleibt. Gleichzeitig sichert das Vorhandensein elementarer Operationen die Möglichkeit, auch zukünftige Strukturierungsmechanismen abzubilden.

In der Programmiersprache ABAP/4 tritt eine ähnliche Problematikauf , wie im letzten Absatz in Zusammenhang mit C++ besprochen. Zahlreiche im Laufe der Entwicklungszeit dieses Systems von einem Makroassembler zu einer Programmiersprache der vierten Generation hinzugefügten Konzepte tragen zu unklaren semantischen Konstrukten bei. Doch sorgt auch hier der Druck, der durch zahlreiche Altanwendungen ausgeübt wird, dafür, daß sich die Situation weiter verschärft. Und so wird auch bei SAP über neue Lösungen, wie die Erweiterung der Systemarchitektur um die Möglichkeit, Java zu interpretieren, nachgedacht. Man begibt sich jedoch damit auf einen Pfad der potentiellen Inkompatibilitäten zu Altsystemen, und so werden weiterhin teure Migrationspfade zu neuen Systemen notwendig bleiben.

6.2 Implementierung

Diese Arbeit wird durch einen praktischen Teil begleitet. Er umfaßt die Implementierung der in Kapitel 4 vorgestellten Compilerschnittstellen und die prototypische Realisierung der darauf aufbauenden Modulverwaltung als ein Beispiel für ein Programmierwerkzeug. Die Modulverwaltung ist als Teil einer Entwicklungsumgebung zu sehen.

Kapitel 5 beschreibt die in die Sprache TLMIN eingebauten Strukturierungsmechanismen. Davon wurden die folgenden Konstrukte vollständig implementiert:

- abgeschlossener Sichtbarkeitsbereich
- selektiv eingeführte Bezeichner
- Umgebungen

Anhang E legt einige Details dieser Implementierung dar. Daraus wird deutlich, daß für die Realisierung dieser auf Programmiersprachenebene angesiedelten Änderungen Komponenten auf mehreren Schichten des TLMIN Systems betroffen sind. Dazu gehört insbesondere der Typüberprüfer, der dergestalt erweitert ist, daß die auf Sichtbarkeitsbereiche bezogenen Operationen semantisch überprüft und anschließend auf vorhandene Elemente des attribuierten Syntaxbaums abgebildet werden können. Die Realisierung von Umgebungen erfordert neben der Bereitstellung einer entsprechenden Programmierschnittstelle und deren Implementierung die Modifikation der Zwischencode-Generierung, um die Anforderungen besonders im Hinblick auf das Zusammenspiel mit den Compilerschnittstellen zu erfüllen. Zu diesen Anforderungen zählt z.B. die schrittweise (Teil-) Übersetzung von Ausdrücken in Umgebungen.

6.3 Ausblick

In dieser Arbeit liegt der Schwerpunkt auf Strukturierungskonzepten und hier besonders auf dem Aspekt der Sichtbarkeit von Namen. Abschnitt 5.2.3 hat angedeutet, daß neben der Betrachtung dieser statischen Aspekte auch dynamische Gesichtspunkte eine Rolle spielen. So wäre in nachfolgenden Arbeiten zu untersuchen, ob und welche Mechanismen zur Unterstützung z.B. von Bindungskonzepten erforderlich sind. Es existieren in TLMIN bislang noch keine ausreichenden sprachlichen und systemtechnischen Mittel, um z.B. alle Techniken der Objektorientierung wie dynamische Methodensuche (*dynamic dispatch*) zu formulieren. Auf dieser Ebene ist weitere Arbeit zu leisten. Vorstellbar ist ein ähnliches Vorgehen wie in dieser Arbeit, so daß ausgehend von Basismechanismen höhere Konzepte realisiert werden können. Diese könnten in einem abschließenden Schritt mit den Ergebnissen dieser Arbeit kombiniert werden.

Auf der Ebene der Entwicklungsumgebung müssen in der konkreten Implementierung des TLMIN Systems inbesondere Werkzeuge zur Unterstützung der Teamentwicklung bereitgestellt werden. So muß es z.B. Werkzeuge zum Verwalten von Softwareeinheiten zwischen in einem Netzwerk verteilten Objektsystemen geben, wenn man davon ausgeht, daß ein Objektsystem als Entwicklungsbereich für jeweils einen Entwickler fungiert. Alternativ müßte die vorliegende Objektspeichertechnologie um die Eigenschaft der Multiuserfähigkeit erweitert werden, wenn davon ausgegangen wird, daß mehrere Entwicklungsbereiche in einem Objektsystem und -speicher zusammengefaßt werden. In dieser Arbeit wird nur die sprachliche Grundlage für die Teamentwicklung gelegt. Auch die Auswirkungen bei Einbeziehung mehrerer kooperierender Objektsysteme muß in diesem Zusammenhang untersucht werden. In diesem Bereich angesiedelte Arbeiten sind [Ramme 1997] und [Schröder 1997].

Auch bei dem dieser Arbeit zugrundegelegten TLMIN System und dessen Entwicklungswerkzeugen bleibt eine Reihe weiterer Arbeiten zu leisten. Die Ausstattung mit Werkzeugen wie z.B. einem Debugger oder einem grafischen Frontend kann mit Hilfe der Compilerschnittstelle in Kombination mit Umgebungen erreicht werden.

Anhang A

Vereinfachte Compilerschnittstelle

In diesem Abschnitt ist die Schnittstelle für die vereinfachte Compilerschnittstelle dargestellt.

```
interface Compiler
 (* System : Tycoon Compiler TLMin
   File   : Compiler.ti
   Author : CK
   Date   : 12-NOV-1996
   Purpose: Interface to the TLMin Compiler
   Update :
*)
export
  Switches <:Ok
  Log <: Ok
  Source <:Ok
  Grammar <:Ok
  Parser <:Ok
  Environment <:Ok
  InitialEnvironment <:Ok
  Context <:Ok
  Bindings <:Ok
  SyntaxTree <:Ok
  TMLCode <:Ok
  TVMCode <:Ok
  Result <:Ok

  contextError :Exception end
  parseError :Exception end
  generateCodeError :Exception end
```

```
 (* functions to manipulate the compiler switches *)
(* switches used by type checker *)
  setTypeCheck (context :Context value :Bool) :Ok
  setTraceSubType (context :Context value :Bool) :Ok
  setTraceType (context :Context value :Bool) :Ok
  setTraceValue (context :Context value :Bool) :Ok
  setTraceApply (context :Context value :Bool) :Ok
  setPrintIndices (context :Context value :Bool) :Ok
  setPrintBackendIndices (context :Context value :Bool) :Ok
  setCacheSubTyping (context :Context value :Bool) :Ok
  setCacheLocalSubTyping (context :Context value :Bool) :Ok
  setCacheExposing (context :Context value :Bool) :Ok

(* switches used by tml-code-generation *)
  setExpandTuningPos (context :Context value :Int) :Ok
  setExpandTuningPosRec (context :Context value :Int) :Ok
  setExpandTuningNeg (context :Context value :Int) :Ok
  setTraceReducer (context :Context value :Bool) :Ok
  setTraceExpander (context :Context value :Bool) :Ok
  setNotifyOptimization (context :Context value :Bool) :Ok

(* switches used by backend *)
  setPrintTVM (context :Context value :Bool) :Ok
  setExecutionLoops (context :Context value :Int) :Ok

(* other switches *)
  setTypedOutput (context :Context value :Bool) :Ok
  setTypedOutputDepth (context :Context value :Int) :Ok

  sourceFromFile(name :String) :Source
  sourceFromString(s :String) :Source
  newContext() :Context
  initContext(context :Context) :Ok
  getGrammar(context :Context) :Grammar
  printGrammarEnv(context :Context) :Ok
  printGrammar(context :Context) :Ok
  getParser(context :Context) :Parser
  getEnvironment(context :Context) :Environment
  parse(context :Context src :Source) :Bindings
  typeCheck(context :Context bindings :Bindings) :SyntaxTree
  printSyntaxTree(context :Context syntaxTree :SyntaxTree) :Ok
  translate(context :Context syntaxTree :SyntaxTree) :TMLCode
  optimize(context :Context tmlCode :TMLCode) :TMLCode
```

```
  printTMLCode(context :Context tmlCode :TMLCode) :Ok
  generateCode(context :Context tmlCode :TMLCode) :TVMCode
  execute(context :Context tvmCode :TVMCode) :Result
  printResult(context :Context result :Result) :Ok
  printResultValue(context :Context result :Result) :Ok
  printLog(context :Context) :Ok
  printLogClear(context :Context) :Ok
  clearLog(context :Context) :Ok
end;
```

Anhang B

Detaillierte Compilerschnittstelle

In diesem Abschnitt ist die Schnittstelle für die detaillierte Compilerschnittstelle dargestellt.

```
interface TLCompiler
(* System : Tycoon Compiler TLMin
   File   : TLCompiler.ti
   Author : CK
   Date   : 27-NOV-1996
   Purpose: low level interface to the TLMin Compiler
   Update :
*)
import
  errorLog :ErrorLog
  environment :Environment
export
  Switches <:Ok
  Let List = Oper(A <:Ok) Ok
  Let Position = Tuple
    where :String
    line, column :Int
  end
  noWhere :Position
  Let Source = Tuple
    var current :Fun() :Char
      (* Return current character, initially " " *)
    var next :Fun() :Ok
      (* Advance current character position.
        An exception will be raised
        if reading past of input *)
    var position :Fun() :Position
      (* Current source position *)
```

```
    var close :Fun() :Ok
      (* Terminate input fromthis source *)
  end
  Let Identifier = Tuple
    pos :Position
    var name :String
  end
  Let Identifiers = List(Identifier)
  Value <:Ok
  Type <:Ok
  State <:Ok
  CheckedBindings <:Ok
  Bindings <:Ok
  wrongBindings :Bindings
  Grammar <:Ok
  EGrammar <:Ok
  Parser <:Ok
  InitialEnvironment <: Ok
  Let Context = Tuple
    switches :Switches
    var src :Source
    errLog :errorLog.T
    var gram :Grammar
    var parser :Parser
    var initialEnvironment :InitialEnvironment
    var env :environment.T
  end
  wrongSyntaxTree :environment.SyntaxTree
grammarError :Exception end
parseError :Exception end
typeCheckError :Exception end
translateError :Exception end
generateCodeError :Exception end

(* functions to manipulate the compiler switches *)
(* switches used by type checker *)
setTypeCheck (switches :Switches value :Bool) :Ok
setTraceSubType (switches :Switches value :Bool) :Ok
setTraceType (switches :Switches value :Bool) :Ok
setTraceValue (switches :Switches value :Bool) :Ok
setTraceApply (switches :Switches value :Bool) :Ok
setPrintIndices (switches :Switches value :Bool) :Ok
setPrintBackendIndices (switches :Switches value :Bool) :Ok
```

```
setCacheSubTyping (switches :Switches value :Bool) :Ok
setCacheLocalSubTyping (switches :Switches value :Bool) :Ok
setCacheExposing (switches :Switches value :Bool) :Ok

(* switches used by tml-code-generation *)
setExpandTuningPos (switches :Switches value :Int) :Ok
setExpandTuningPosRec (switches :Switches value :Int) :Ok
setExpandTuningNeg (switches :Switches value :Int) :Ok
setTraceReducer (switches :Switches value :Bool) :Ok
setTraceExpander (switches :Switches value :Bool) :Ok
setNotifyOptimization (switches :Switches value :Bool) :Ok

(* switches used by backend *)
setPrintTVM (switches :Switches value :Bool) :Ok
setExecutionLoops (switches :Switches value :Int) :Ok

(* other switches *)
setTypedOutput (switches :Switches value :Bool) :Ok
setTypedOutputDepth (switches :Switches value :Int) :Ok

sourceFromFile(name :String) :Source
sourceFromString(s :String) :Source
newSwitches() :Switches
newLog() :errorLog.T
newGrammar() :Grammar
getEGrammar(gram :Grammar) :EGrammar
printGrammarEnv(gram :Grammar) :Ok
printGrammar(gram :Grammar) :Ok
newParser(gram :Grammar) :Parser
newInitialEnvironment() :InitialEnvironment
newEnvironment() :environment.T
initEnvironment(initEnv :InitialEnvironment env :environment.T
                log :errorLog.T) :InitialEnvironment
printEnvironment(env :environment.T) :Ok
parse(var parser :Parser var gram :Grammar
      src :Source log :errorLog.T) :Bindings
typeCheck(switches :Switches
          initialEnvironment :InitialEnvironment
          env :environment.T
          bindings :Bindings
          log :errorLog.T) :environment.SyntaxTree
```

```
printSyntaxTree(switches :Switches
                syntaxTree :environment.SyntaxTree
                log :errorLog.T) :Ok
printSignature(switches :Switches
               signature :environment.Signature
               log :errorLog.T) :Ok
translate(switches :Switches
          initialEnvironment :InitialEnvironment
          env :environment.T
          syntaxTree :environment.SyntaxTree
          log :errorLog.T) :environment.TMLCode
optimize(switches :Switches
         tmlCode :environment.TMLCode
         log :errorLog.T) :environment.TMLCode
printTMLCode(tmlCode :environment.TMLCode
             log :errorLog.T) :Ok
generateCode(switches :Switches
             env :environment.T
             tmlCode :environment.TMLCode
             log :errorLog.T) :environment.TVMCode
execute(switches :Switches
        env :environment.T
        tvmCode :environment.TVMCode
        log :errorLog.T) :environment.Result
printResult(switches :Switches
            result :environment.Result
            log :errorLog.T) :Ok
printResultValue(result :environment.Result
                 log :errorLog.T) :Ok
saveSystem(log :errorLog.T) :Ok
printLog(log :errorLog.T) :Ok
printLogClear(log :errorLog.T) :Ok
clearLog(log :errorLog.T) :Ok
end;
```

Anhang C

Schnittstelle zu Umgebungen

In diesem Abschnitt ist die Schnittstelle für Umgebungen dargestellt. Sie kann durch Funktionsaufrufe oder nach Syntaxerweiterung durch die Verwendung entsprechender Schlüsselwörter benutzt werden.

Sie entspricht nicht der innerhalb des TLMIN Systems verfügbaren Schnittstelle *Environment*. Dies ist darin begründet, daß der hier abstrakt gehaltene Typ für Umgebungen in der systeminternen Variante offengelegt ist, um insbesondere der Implementierung der Compilerschnittstelle Zugriff darauf zu geben.

```
interface EnvironmentReflect
 (* System : Tycoon Compiler TLMin
   File    : Environment.ti
   Author  : CK
   Date    : 09-JAN-1997
   Purpose: interface to tlmin-Environments uses for reflection
   Updates:
*)

import
  errorLog
  iter

export
  T <:Ok
  Signature <:Ok
  Binding <:Ok
  SyntaxTree <:Ok
  TMLCode <:Ok
  TVMCode <:Ok
  Result <:Ok
  new() :T
```

```
  names(environment :T) :iter.T(String)
  nameOfBinding(binding :Binding) :String
  signatureOfBinding(binding :Binding) :Signature
  printValueOfBinding(binding :Binding) :Ok
  printRuntimeValueOfBinding(binding :Binding) :Ok
  printOIDValueOfBinding(binding :Binding) :Ok
  getBinding(environment :T  name :String
             log :errorLog.T) :Binding
  addBinding(environment :T  binding :Binding) :T
  deleteBinding(environment :T name :String log
                :errorLog.T) :T
  copyBinding(source :T destination :T name :String log
              :errorLog.T) :T
  copyBindingAs(source :T  destination :T
                from :String  to :String
                log :errorLog.T) :T
  moveBinding(source :T  destination :T
              name :String  log :errorLog.T) :T
  insertSyntaxTree(env :T  syntaxTree :SyntaxTree) :T
  insertTMLCode(env :T  tmlCode :TMLCode) :T
  insertRuntimeValue(env :T  result :Result
                     log :errorLog.T) :T
end;
```

Anhang D

Beispiel zum Rautenimport

Die folgenden Seiten geben das in Abschnitt 5.2.4 eingeführte Beispiel zur Thematik des Rautenimports in voller Länge wieder. Dabei wird angenommen, daß bestimmte für dieses Beispiel nicht relevante Bezeichner, wie z.B. *list*, *iter* usw. sich bereits in der Umgebung befinden, gegen die diese Ausdrücke übersetzt werden.

Das in Abschnitt 5.2.4 verwendete Modul *fastStack* ist hier nicht explizit aufgeführt, da es sich nicht wesentlich von *stack* unterscheidet und an dieser Stelle der Schwerpunkt auf der Verwendung der Sichtbarkeitsoperationen liegen soll. Das Modul *stack* sowie die Schnittstelle *Stack* wird in Abbildung 5.9 auf Seite 73 bzw. Abbildung 5.5 auf Seite 69 aufgeführt.

D.1 Schnittstelle *Kreditor*

```
Let Kreditor = Scope with StdType StdIde Stack do
Oper(Let StdType = StdType()  stdType :StdType
     Let StdIde = StdIde(:StdType) stdIde :StdIde
     Let Stack = Stack(:StdType :StdIde stdIde)  stack :Stack)
  Scope with stack stdIde do
    Tuple
      get():stack.T(:stdIde.Int)
    end
  end
end
```

D.2 Modul *kreditor*

```
let kreditorLINK =
scope with Stack StdType Kreditor do
  fun(Let StdType = StdType()  stdType :StdType
      Let StdIde = StdIde(:StdType) stdIde :StdIde
      Let Stack = Stack(:StdType :StdIde stdIde)  stack :Stack
```

```
      Let Kreditor = Kreditor(:StdType  stdType
                              :StdIde stdIde
                              :Stack  stack)) :Kreditor
  scope with stack stdType Kreditor do
    let get() :stack.T(:stdType.Int) =
      ...
    let result :Kreditor =
    tuple
      let get = get
    end
    result
  end
end
```

D.3 Schnittstelle *Debitor*

```
Let Debitor = Scope with StdType StdIde Stack do
Oper(Let StdType = StdType()  stdType :StdType
     Let StdIde = StdIde(:StdType) stdIde :StdIde
     Let Stack = Stack(:StdType :StdIde stdIde)  stack :Stack)
  Scope with stack stdIde do
    Tuple
      put(a :stack.T(:stdIde.Int)) :stdIde.Bool
    end
  end
end
```

D.4 Modul *debitor*

```
let debitorLINK =
scope with Stack StdType Debitor do
  fun(Let StdType = StdType stdType :StdType
      Let StdIde = StdIde(:StdType) stdIde :StdIde
      Let Stack = Stack(:StdType :StdIde stdIde)  stack :Stack
      Let Debitor = Debitor(:StdType stdType
                            :StdIde stdIde
                            :Stack stack)) :Debitor
  scope with stdType stack Debitor do
    let put(a :stack.T(:stdType.Int)) :stdType.Bool =
      ...
    let result :Debitor =
```

```
    tuple
      let put = put
    end
    result
  end
end
```

D.5 Schnittstelle *Mandant*

```
Let Mandant = Scope with StdType Stack Debitor Kreditor do
Oper(Let StdType = StdType()  stdType :StdType
     Let StdIde = StdIde(:StdType) stdIde :StdIde
     Let Stack = Stack(:StdType :StdIde stdIde)  stack :Stack
     Let Debitor = Debitor(:StdType stdType
                           :StdIde stdIde
                           :Stack stack)  debitor :Debitor
     Let Kreditor = Kreditor(:StdType stdType
                             :StdIde stdIde
                             :Stack stack)  kreditor :Kreditor)
  Scope with stdIde do
    Tuple
      f() :stdIde.Bool
    end
  end
end
```

D.6 Modul *mandant*

```
let mandantLINK =
scope with Stack StdType Debitor Kreditor Mandant do
  fun(Let StdType = StdType  stdType :StdType
      Let StdIde = StdIde(:StdType)
      Let Stack = Stack(:StdType :StdIde stdIde)  stack :Stack
      Let Debitor = Debitor(:StdType stdType
                            :StdIde stdIde
                            :Stack stack)  debitor:Debitor
      Let Kreditor = Kreditor(:StdType stdType
                              :StdIde stdIde
                              :Stack stack)  kreditor :Kreditor
      Let Mandant = Mandant(:StdType stdType
                              :StdIde stdIde
```

```
                            :Stack stack
                            :Debitor debitor
                            :Kreditor kreditor)) :Mandant
  scope with stdIde debitor kreditor Mandant do
    let f() :stdIde.Bool =
      debitor.put(kreditor.get())
    let result :Mandant =
    tuple
      let f = f
    end
    result
  end
end
```

Anhang E

Implementierung

In diesem Teil werden technische Details der Implementierung der in Kapitel 5 vorgestellten Elemente des TLMIN System dargelegt. Zum Verständnis werden vom Leser detaillierte Kenntnisse der TLMIN Architektur und Systemimplementierung vorausgesetzt.

E.1 Sichtbarkeitsbereiche

Die Implementierung der Sichtbarkeitsbereiche, wie in Abschnitt 5.1 beschrieben, umfaßt im einzelnen folgendes:

1. Änderung der Grammatik, so daß die neuen Konstrukte erkannt werden. Dazu wurden gemäß der Produktion einer Sequenz weitere Produktionen eingeführt, die optional Namen (*with* ... *do*) übernehmen. Es ergeben sich somit vier neue Konstruktoren, die die Grammatik erweitern:

   - *mkValueScope*, um einen abgeschlossenen Sichtbarkeitsbereich ohne importierte Bezeichner zu generieren.
   - *mkValueScopeWith*, um in den Sichtbarkeitsbereich Bezeichner aus dem übergeordneten Block zu importieren.
   - *mkTypeScope* schließt den Sichtbarkeitsbereich einer Typdefinition ab.
   - *mkTypeScopeWith* importiert die als Argument übergebenen Bezeichner in den leeren Sichtbarkeitsbereich der Typdefinition.

   Eine Änderung des Parsers ist nicht notwendig. Er wird mit der neuen Grammatik parametrisiert und ruft die neuen Konstruktoren automatisch auf.

2. In die vom Parser erzeugte Darstellung, dem vorläufigen Syntaxbaum (PreAST), der als Eingabe zur Typüberprüfung herangezogen wird, wurden weitere Knotentypen eingefügt. Diese wurden wieder analog einer Sequenz (*begin* ... *end*) gestaltet, mit dem Unterschied, daß die optional zu importierenden Namen enthalten sind. An dieser Stelle werden die vier

möglichen Fälle (s. obige Aufzählung) zu zwei verbleibenden Möglichkeiten normalisiert, indem die zunächst optionale Liste von importierten Bezeichnern obligatorisch wird. Im Falle, daß keine Bezeichner importiert werden, ist die Importliste leer.

3. Der Typüberprüfer wurde modifiziert. Er enthält Code zur Behandlung des neuen Knotens des vorläufigen Syntaxbaumes. Dort wird dafür gesorgt, daß die Überprüfung des in das Sichtbarkeitskonstrukt eingeschlossenen Bereichs gegen eine neue Umgebung erfolgt. Dies wird erreicht, indem die Typüberprüfung des Sichtbarkeitsbereiches an einer veränderten Kopie der aktuellen Typumgebung stattfindet. Diese Typumgebung enthält alle sichtbar gemachten (importierten) Bezeichner an deren bisherigen Positionen und an den übrigen Platzhalterwerten. Diese Vorgehensweise führt dazu, daß, obwohl nur wenige Bezeichner sichtbar sind, dennoch die neue Typumgebung gegenüber der alten unverändert viel Speicherplatz verbraucht. Diese Lösung hat allerdings den Vorteil, keine weiteren Änderungen im Typüberprüfer nach sich zu ziehen, wie z.B. eine Umrechnung von Indizes, die in die Typumgebung zeigen, und die invalidiert wären, wenn die Position der Bezeichner durch Schrumpfung der Typumgebung verändert wird.

   Weiterhin müssen die als Argumente des Sichtbarkeitskonstruktes angegebenen Namen auf ihr Vorhandensein in der umschließenden Umgebung geprüft und zur neuen Typumgebung hinzugefügt werden.

Für das Sichtbarkeitskonstrukt ist keine Laufzeitunterstützung notwendig. Das ist begründet in der Tatsache, daß es sich hierbei um eine rein statische Abprüfung von Sichtbarkeit bzw. Vorhandensein von Bezeichnern handelt, die zur Übersetzungszeit vollständig behandelt werden kann. Der für die innerhalb des Sichtbarkeitskonstruktes liegenden Anweisungen erzeugte Code kann gegen die dort gültige Umgebung ausgeführt werden. Es muß keine neue Laufzeitumgebung geschaffen werden, da sich die referenzierten Bezeichner durch die besondere Behandlung der Typumgebung innerhalb des Sichtbarkeitskonstruktes exakt an der erwarteten Position befinden. Auf nicht sichtbare Bezeichner, die sich zur Ausführungszeit auch in der Laufzeitumgebung des betreffenden Codes befinden, kann nicht zugegriffen werden. Dies ist gerade durch die Typüberprüfung statisch erzwungen worden.

E.2 Umgebungen

Das in TLMIN verwendete Modell der Umgebungen ist ausführlich in Abschnitt 5.3 beschrieben. An dieser Stelle wird auf die konkreten Aspekte der Implementierung eingegangen.

E.2.1 Aufbau

Umgebungen werden im Übersetzer von der Phase der Typüberprüfung bis zur Ausführung des Codes der virtuellen Maschine in der Tycoon Maschine benutzt. Sie stellen ergänzend zu den verschiedenen Zwischen- und Coderepräsentationen die Verbindung zwischen dem Compiler als statische Komponente einerseits und den aktuellen dynamischen Daten (Programme, Ausdrücke) andererseits her.

Eine Umgebung setzt sich aus drei Teilkomponenten zusammen. Jede dieser Teilkomponente ist eine Massendatenstruktur, die Verweise auf die eigentliche Information enthält:

Typumgebung: Sie umfaßt die Signaturen aller sichtbaren Bezeichner wie Funktionen, Variablen etc. Unter Signaturen wird in diesem Zusammenhang die Repräsentation dieser Signaturen im Syntaxbaum verstanden. Konkret sind in der Typumgebung somit Verweise auf Teilsyntaxbäume enthalten.

Wertumgebung: Die Codegenerierung übersetzt die Repräsentation des Syntaxbaums in die Darstellung als TML-Baum. Die Verweise auf die entstandenen Knoten werden in der Wertumgebung abgelegt.

Laufzeitumgebung: Nach Erzeugung und Ausführung des TVM-Codes werden die evaluierten Werte in die Laufzeitumgebung eingetragen und stehen damit nachfolgenden Ausführungsschritten zur Verfügung.

Die Operationen zum Erzeugen von Umgebungen und zum Auffinden von Komponenten lassen sich leicht implementieren, da sie größtenteils auf bereits vorhandenem Code aufbauen können. Sie werden hier nicht weiter behandelt. Die Aufmerksamkeit soll stattdessen auf das Kopieren und Verschieben von Komponenten zwischen Umgebungen gelenkt werden.

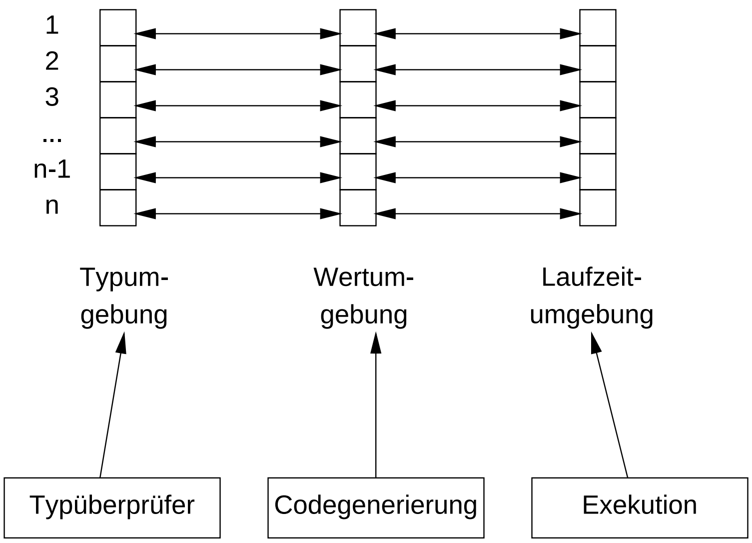


Abbildung E.1: Beziehungen der drei Teilkomponenten einer Umgebung.

Die bisherige Implementierung der drei Einzelkomponenten der Umgebungen sind dergestalt realisiert, daß sie von einer strengen Abfolge des Übersetzungsvorganges ausgehen. Konkret heißt das z.B., daß die Einträge der Wertumgebung mit den Einträgen der Typumgebung korrespondieren, weil die durch den Syntaxbaum implizit vorgegebene Reihenfolge durch den

Algorithmus der Codegenerierung erhalten bleibt (s. Abbildung E.1). Überlappende Aufrufe der Compilerfunktionen würden zu inkonsistenten Umgebungen führen. Beim Anlegen neuer Einträge in einer Umgebungskomponente muß also nicht nach einem freien Platz, sondern nach dem korrespondierenden Eintrag gesucht werden.

Ein weiteres Problem bei der Implementierung von Umgebungen ergab sich aus der Anforderung, Elemente zwischen Umgebungen frei kopieren und verschieben zu können. Die in der Umgebung abgelegten Strukturen beziehen sich teilweise auf Einträge in derselben Umgebung. Diese Verweise sind in der Implementierung des TLMIN Systems im Gegensatz zur Implementierung in einem TL System nicht über de Bruijn Indizes gelöst, sondern über direkte Verweise [Bremer 1996]. Dies erleichtert es grundsätzlich, ein Kopieren zwischen Umgebungen zu realisieren, da die enthaltenen Strukturen sich nicht auf die Umgebung beziehen, sondern direkt auf die in ihnen enthaltenen Strukturen. Es gibt jedoch Ausnahmen in der Implementierung des TLMIN Systems, die weiterhin über Indizes modelliert sind. Diese betreffen ausschließlich den Typüberprüfer bzw. die von ihm aufgebaute Struktur des Syntaxbaums. Dies sind im einzelnen:

- Bei der Definition eines Tupelfeldes bezeichnet ein Index die Position des Feldes im entsprechenden Tupeltyp. Da bei Rekordfeldern die Feldidentifikation per Namen geschieht, tritt in diesem Fall kein Index auf.

  Die Verwendung dieses Indexes bezieht sich nicht auf die Umgebung direkt, sondern auf die Position innerhalb einer Struktur, die in der Umgebung abgelegt ist. Für das oben geschilderte Problem spielt dieser Index daher keine Rolle.

- Die Einträge der Typumgebung verweisen auf Signaturen, die wiederum auf andere Komponenten wie Werte, Bindungen oder Typen verweisen können. Jede Signatur besitzt eine Komponente, die die Position in der Typumgebung angibt. Da es sich hier um einen Index handelt, der sich auf eine Umgebung bezieht, muß er für oben genanntes Problem berücksichtigt werden. Zwar wird in der Implementierung des TLMIN Systems dieser Index nur zu Informationszwecken verwendet, diese Information sollte jedoch weiterhin zur Verfügung stehen. Dabei ist zu beachten, daß diese Verweisinformation sich nun auch auf andere Umgebungen beziehen kann. Ein Index reicht zur Identifikation deshalb nicht aus.

- Auch bei der Repräsentation von Werten im Syntaxbaum treten Indizes auf. Neben der Benutzung bei Feldindizierungen und zum Auffinden einer Variante eines varianten Tupeltyps ist vor allem die Verwendung von Bezeichnern interessant, die sich auf zuvor definierte Variablen beziehen. Hier werden Indizes eingesetzt, um auf die Position der referenzierten Typrepräsentation in der Umgebung zu verweisen.

  Die Verwendung dieses Indexes ist im Sinne des oben beschriebenen Problems kritisch, da ein derartiger Index in einer anderen Umgebung keine Gültigkeit mehr besitzt.

Lösungsansätze

Grundsätzlich besteht das Problem, daß Verweise, in diesem Fall als Index in einer Umgebung realisiert, nur lokale Gültigkeit besitzen. Der Gültigkeitsbereich dieser Verweise soll im Rahmen der Implementierung der Umgebungen umgebungsübergreifend gestaltet werden. Dazu bieten sich mehrere Ansätze an:

1. Der Index könnte um eine zusätzliche Komponente erweitert werden, die angibt, in welcher Umgebung die Verweisinformation gültig ist. Bei dieser Lösung kann das Problem der hängenden Referenzen (*dangling reference*) auftreten, wenn Einträge aus einer Umgebung gelöscht werden und nicht dafür gesorgt wird, daß alle Verweise auf das zu löschende Element eliminiert werden. Hier tritt dann eventuell ein weiteres Problem auf, da gelöschte Komponenten nicht mit dem neuen Verweisschema referenziert werden können, wenn sie sich in keiner Umgebung mehr befinden. Es muß zur Lösung eine weitere Umgebung gehalten werden, die bereits gelöschte Einträge aufnimmt.

2. Die Datenstruktur des Syntaxbaumes könnte in der Repräsentation für Werte so geändert werden, daß direkt auf die referenzierte Typinformation verwiesen wird und nicht auf einen Eintrag in der Umgebung. Bei diesem Lösungsansatz ergibt sich die Schwierigkeit, daß zum Zeitpunkt der Erstellung des Indexes die referenzierte Information noch nicht vorliegt. Es muß mit einer variablen Bindung gearbeitet werden, die aktualisiert wird, sobald die referenzierte Signatur vorliegt.

3. Da der Index in der Datenstruktur für Werte bei der Codegenerierung ausschließlich dafür verwendet wird, den entsprechenden Eintrag in der Wertumgebung zu lokalisieren, bietet sich eine weitere Lösungsmöglichkeit an. Das Auffinden kann auf Basis des Namens des entsprechenden Bezeichners geschehen. In diesem Fall dient die Namensinformation als Verweis. Zu beachten ist hierbei, daß Bezeichner sowohl in unterschiedlichen Blöcken als auch im gleichen Block mehrfach verwendet werden können. Beim Auffinden ist dies zu berücksichtigen.

4. Die Analyse der auftretenden Indizes hat ergeben, daß diese nur in der Repräsentation von Werten auftreten und nur auf die Signaturen der entsprechenden Wertdefinition verweisen[1]. Es können nun in einer globalen Massendatenstruktur Verweise auf alle angelegten Signaturen für Werte gespeichert werden. Der Verweis auf diese Wertsignaturen erfolgt über einen entsprechenden Schlüsselwert des Eintrages oder über einen Index bei der Realisierung mit Hilfe einer indizierten Struktur.

E.2.2 Realisierung

Wie in Punkt vier des letzten Abschnittes dargelegt, treten Verweise auf Werte in der Wertkomponente einer Umgebung nur in Form von Verweisen auf bestimmte Objekte auf. Eine systemweite Verwaltung aller Einträge in Wertkomponenten generiert bei jeder Erzeugung eines neuen Objektes, auf das verwiesen werden kann, einen eindeutigen Index für dieses Objekt. Dieser Index wird zusätzlich im Syntaxbaum gespeichert.

[1] Indizes treten wie beschrieben auch an anderer Stelle auf, haben dort aber nur Informationscharakter. Aus diesem Grund werden sie hier nicht in die Betrachtung einbezogen. Es wird davon ausgegangen, daß die mit Hilfe dieser Indizes erstellten Fehlermeldungen auch auf andere Weise erzeugt werden können

In der Zwischencode-Generierung, in der der Syntaxbaum abgewandert wird, wird anhand des zuvor dort gespeicherten Indexes der Werteintrag in der systemweiten Verwaltung der Werteinträge abgelegt. Auf diese Weise haben alle anderen Übersetzungsvorgänge Zugriff auf die Werteinträge anderer Umgebungen.

Literaturverzeichnis

AG : AG, SAP. „ABAP/4 Online-Help“.

Aho et al. 1988: Aho, Alfred V., Sethi, Ravi, und Ullman, Jeffrey D. *Compilerbau, Teil I*. Addison-Wesley, 1988.

Balzert 1982: Balzert, Helmut. *Die Entwicklung von Software-Systemen: Prinzipien, Methoden, Sprachen, Werkzeuge*. Informatik. Bibliographisches Institut, Mannheim, Wien, Zürich, 1982.

Boehm 1986: Boehm, Barry W. „A Spiral Model of Software Development and Enhancement“. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, August 1986.

Booch 1994: Booch, Grady. *Objektorientierte Analyse und Design*. Addison-Wesley, 1994.

Bremer 1996: Bremer, Gerd. „Typüberprüfung in polymorphen Programmiersprachen: Aufgaben und Lösungsansätze“. Diplomarbeit, Arbeitsbereich Datenbanken und Informationssysteme, Fachbereich Informatik, Universität Hamburg, August 1996.

Brooks 1995: Brooks, Frederick P. *The Mythical Man-Month: Essay on Software Engineering*. Addison Wesley, Jubiläums- Auflage, 1995.

Brown et al. 1992: Brown, Alan W., Earl, Anthony N., und McDermid, John A. *Software Engineering Environments: Automated Support for Software Engineering*. International series in software engineering. McGraw-Hill, London u.a., 1992.

Cin et al. 1986: Cin, Mario Dal, Lutz, Joachim, und Risse, Thomas. *Programmierung in Modula-2*. B. G. Teubner, zweite Auflage, 1986.

Comer 1997: Comer, Edward R. „Alternative Software Life Cycle Models“. In: *Software engineering*, Seite 404–414. IEEE Computer Society Press, 1997.

Dearle 1989: Dearle, A. „Environments: A flexible binding mechanism to support system evolution“. In: *Proc. 22nd International Conference on System Science*, Seite 46–55, Hawaii, 1989.

Duden 1988: *Duden Informatik: Sachlexikon für Studium und Praxis*. Bibliographisches Institut & F. A. Brockhaus AG, Mannheim, Wien, Zürich, 1988.

Feldmann 1979: Feldmann, Stuart I. „Make – A Program for Maintaining Computer Programs“. *Software – Practice & Experience*, 9:255–265, 1979.

Gawecki und Matthes 1996: Gawecki, Andreas und Matthes, Florian. „Exploiting Persistent Intermediate Code Repressentations in Open Database Environments“. In: *Proceedings of 5th Conference on Extending database Technology, EDBT ’96, Avignon France*. Universität Hamburg, März 1996.

Geisler 1995: Geisler, Andreas. „Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache“. Diplomarbeit, Universität Hamburg, Juni 1995.

III 1995: III, Thomas A. Curran. „Application Development Tools for Client/Server Computing: ABAP/4 Development Workbench“. Technischer Bericht, TCManagement Inc., Januar 1995.

Kernighan und Richie 1990: Kernighan, Brian W. und Richie, Dennis M. *Programmieren in C*. Carl Hanser Verlag, zweite Auflage, 1990.

Kiradjiev 1994: Kiradjiev, Plamen. „Dynamische Optimierung in CPS-orientierten Zwischensprachen“. Diplomarbeit, Arbeitsbereich Datenbanken und Informationssysteme, Fachbereich Informatik, Universität Hamburg, Dezember 1994.

Koch 1996a: Koch, Christian. „Einführung in die ABAP/4-Entwicklungsumgebung“. Technischer Bericht, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Datenbanken und Informationssysteme, 1996.

Koch 1996b: Koch, Christian. „Generisches und typsicheres Modulmanagement in Tycoon“. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich DBIS, 1996.

Louden 1994: Louden, Kenneth C. *Programmiersprachen Grundlagen, Konzepte, Entwurf*. Thomson Publishing, erste Auflage, 1994.

Mathiske et al. 1993: Mathiske, Bernd, Matthes, Florian, und Müßig, Sven. „The Tycoon System and Library Manual“. Technischer Bericht DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

Matthes und Müßig 1993: Matthes, Florian und Müßig, Sven. „The Tycoon Language TL: An Introduction“. Technischer Bericht DBIS Tycoon Report 112-93, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

Matthes und Schmidt 1991: Matthes, Florian und Schmidt, Joachim W. „Bulk Types: Built-In or Add-On?“. In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Greece*. Morgan Kaufman Publishers, September 1991.

Matthes und Schmidt 1992: Matthes, Florian und Schmidt, Joachim W. „Definition of the Tycoon Language TL – A Preliminary Report“. Technischer Bericht Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

Matthes 1993: Matthes, Florian. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer Verlag, 1993.

Matzke 1996: Matzke, Bernd. *ABAP/4: Die Programmiersprache des SAP-Systems R/3*. Edition SAP. Addison Wesley, Bonn u.a., 1996.

Meyer 1988: Meyer, Bertrand. *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag, 1988.

Microsystems : Microsystems, Sun. „Teamware Answerbook“.

Morrison et al. 1988: Morrison, R., Brown, A.L., Carrick, R., Connor, R., und Dearle, A. „The Napier Reference Manual“. Reference manual, Univerity of St Andrews, St Andrews, Schottland, 1988.

Morrison et al. 1993: Morrison, R., Baker, C., Conner, R. C. H., Cutts, Q. I., und Kirby, G. N. C. „Approaching Integration in Software Environments“. Technischer Bericht CS/93/10, University of St. Andrews, 1993.

Morrison et al. 1994: Morrison, R., Connor, R. C. H., Cutts, Q. I., und Kirby, G. N. C. „Persistent Possibilities for Software Environments“. In: *The Intersection between Databases and Software Engineering*, Seite 78–87. IEEE Computer Society Press, 1994.

NIST 1993: NIST, ECMA. „Reference Model for Frameworks of Software Engineering Environments“. Technischer Bericht NIST Special Publication 500-211, NIST ISEE National Institute of Standards and Technology United States Department of Commerce and ECMA European Computer Manufacturers Association TC33 Task Group on the Reference Model, August 1993.

Nye 1993a: Nye, Adrian. *Xlib Programming Manual for Version 11*, volume 1 of *The Definitive Guides to the X Window System*. O'Reilly & Associates, Inc, dritte Auflage, 1993.

Nye 1993b: Nye, Adrian, Hrsg. *Xlib Reference Manual for Version 11*, volume 2 of *The Definitive Guides to the X Window System*. O'Reilly & Associates, Inc, dritte Auflage, 1993.

Pakendorf 1996: Pakendorf, Martin. „Optimierung der persistenten objektorientierten Programmiersprache TOOL“. Diplomarbeit, Arbeitsbereich Datenbanken und Informationssysteme, Fachbereich Informatik, Universität Hamburg, Juni 1996.

Ramme 1997: Ramme, Kay. „Dynamisches Rebinden von Objekten in kooperierenden persistenten Objektsystemen“. Diplomarbeit, Arbeitsbereich Datenbanken und Informationssysteme, Fachbereich Informatik, Universität Hamburg, Juli 1997.

Reifer 1993: Reifer, D. J. „Software Management“. Technischer Bericht, IEEE, 1993.

Rochkind 1975: Rochkind, Marc J. „The Source Code Control System“. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, 1975.

Royce 1970: Royce, W. W. „Managing the Development of Large Software Systems: Concepts and Techniques“. *WESCON Technical Papers*, 14, 1970.

SAP 1994: SAP AG. *Funktionen im Detail – ABAP/4 DEVELOPMENT WORKBENCH*, September 1994.

Schröder 1994: Schröder, Gerald. „Syntaktische Erweiterbarkeit von Programmiersprache unter Benennungs-, Bindungs- und Typisierungsinvarianzen“. Diplomarbeit, Arbeitsbereich Datenbanken und Informationssysteme, Fachbereich Informatik, Universität Hamburg, Februar 1994.

Schröder 1997: Schröder, Gerald. *Persistente kooperierende Objektsysteme*. Dissertation, Arbeitsbereich Datenbanken und Informationssysteme, Fachbereich Informatik, Universität Hamburg, 1997.

Sommerville 1995: Sommerville, Ian. *Software Engineering*. Addison–Wesley, fünfte Auflage, 1995.

Stroustrup 1991: Stroustrup, Bjarne. *The C++ Programming Language*. Addison Wesley, zweite Auflage, 1991.

van der Veer et al. 1988: Veer, G. van der, Green, T. R., Hoc, J-M, und D. *Working with Computers: Theory versus Outcome*. Academic Press, 1988.

Will et al. 1995: Will, Liane, Hienger, Christiane, Straßenburg, Frank, und Himmer, Rocco. *R/3-Administration*. Addison-Wesley, 1995.

Wirth 1979: Wirth, Niklaus. „The Module: A System Structuring Facility in High-Level Programming Languages“. In: Tobias, Jeffrey M., Hrsg., *Language Design and Programming Methodology*, Seite 1–24. Springer Verlag, September 1979.

Erklärung

Ich versichere hiermit, die vorliegende Arbeit selbständig und ausschließlich unter Zuhilfenahme der angegebenen Quellen durchgeführt zu haben.

Ort, Datum   Christian Koch