Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Wintersemester 1995/96

Studienarbeit

Generisches und typsicheres Modulmanagement in Tycoon

vorgelegt von:
Christian Koch
Norderstedt

Betreuer:
Gerald Schröder
Prof. Dr. Joachim W. Schmidt

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 2
  - 1.1.1 Separate Übersetzung . . . 4
  - 1.1.2 Kapselung . . . 4
  - 1.1.3 Hierarchische Strukturierung . . . 5
  - 1.1.4 Unabhängige Entwicklung . . . 5
- 1.2 Einordnung . . . 5
  - 1.2.1 Objektidentifikation . . . 6
  - 1.2.2 Systemgenerierung . . . 6

2 Objekte der Tycoon Modulverwaltung **9**
- 2.1 Schnittstelle . . . 9
- 2.2 Modul . . . 9
- 2.3 Bibliothek . . . 12
- 2.4 Versteckte Bezeichner . . . 13
- 2.5 Zustand . . . 14
  - 2.5.1 Modul . . . 15
  - 2.5.2 Schnittstelle . . . 15

3 Umgebung der Tycoon Modulverwaltung **17**
- 3.1 Objektspeicher . . . 18
- 3.2 Externer Speicher . . . 18
- 3.3 Fingerprint . . . 19
- 3.4 Toplevel . . . 19
- 3.5 Wurzelbibliothek . . . 20

4 Aufgaben der Tycoon Modulverwaltung **21**
- 4.1 Übersetzen . . . 22
- 4.2 Binden und Definieren . . . 22
- 4.3 Sekundärspeicherzugriff . . . 22
- 4.4 Versionierung und Konfigurationsmanagement . . . 23
- 4.5 Zusammenfassung . . . 23

5 Semantik der Toplevel-Kommandos **25**
- 5.1 import *components* . . . 25
- 5.2 do makeComponent *component* . . . 26
- 5.3 do make *component* . . . 27

5.4 do makeLibraries *component* 27
5.5 do updateRootLibrary *component* 27
5.6 do loadComponent *component* 28
5.7 do link *component* 28
5.8 do unlink *component* 28
5.9 Bewertung 29

6 Implementierung der Tycoon Modulverwaltung 31
6.1 Architektur 31
6.1.1 tlObject 32
6.1.2 tyDepend 33
6.1.3 tlLibrary 33
6.1.4 Schnittstelle zur Modulverwaltung 33
6.2 Abläufe 34
6.2.1 Importieren 34
6.2.2 Übersetzung 39

7 Erweiterungen und Verbesserungen 41
7.1 Link-Vektor 41
7.2 Übersetzen von Bibliotheken 42
7.3 Übersetzen von versteckten Komponenten 43
7.4 Verzeichnis aller geladenen Komponenten 44
7.5 Synchronisieren des Objektspeichers 45

8 Generische und typsichere Systemgenerierung 47
8.1 Ansätze 48
8.2 UNIX-Make 49
8.2.1 Regeln 49
8.2.2 Implizite Regeln 49
8.2.3 Abhängigkeiten und Datenfluß 50
8.2.4 Automatische Abhängigkeiten 51
8.3 Systemgenerierung in TL 51
8.3.1 Repräsentation 52
8.3.2 Fehlerbehandlung 53
8.4 Benutzung 54
8.5 Bewertung 55
8.6 Ausblick 56

A Schnittstelle des Moduls *make* 59

Literaturverzeichnis 61

Kapitel 1

Einleitung

Diese Arbeit beschaftigt sich mit der Modulverwaltung des Tycoon Systems. Ziel ist es, die existierende Modulverwaltung von Tycoon, die aus einem ad hoc-Ansatz heraus entstanden und weitgehend undokumentiert ist, zu analysieren. Konzepte und Aufgaben, die umgesetzt bzw. nicht umgesetzt wurden, werden betrachtet. Auf diesen Betrachtungen liegt der Schwerpunkt dieser Arbeit. Sie bilden außerdem die Grundlage für nachfolgende Arbeiten, aber auch Verbesserungen der Performanz können hiervon abgeleitet werden. Die bislang fehlende und fehlerhafte Dokumentation wird mit dieser hinzugefügt bzw. ersetzt fehlerhafte. Bekannte Fehler werden korrigiert, unbekannte Fehler durch systematische Tests gesucht und unvollständig implementierte Merkmale bzw. in ihrer Semantik unklare Eigenschaften (*hide*-Klausel, geschachtelte Bibliotheken) werden einer kritischen Betrachtung unterzogen.

Durch Analyse von häufig gebrauchten Datenstrukturen und durchlaufenen Funktionen können Engpässe lokalisiert und durch alternative Datenstrukturen ersetzt werden. Andere Änderungen erfordern es, Algorithmen neu zu entwerfen, zu implementieren und zu bewerten.

Schließlich ist ein weiteres Ziel, die gewonnenen Kenntnisse anzuwenden, die wesentlichen Konzepte der Modulverwaltung zu erfassen und vom Rest des Tycoon Compilers zu isolieren. Auf diese Weise kann ein möglichst weitgehend wiederverwendbarer *Modulmanager* entstehen. Hierzu wird mit der Entwicklung des generischen Make ein Schritt getan. Weitere Schritte in diese Richtung müssen noch folgen. Ganz allgemein ist zu untersuchen, ob die Aufgabe der Verwaltung interner Objekte im Objektspeicher und externer Objekte auf Sekundärspeicher in einer generischen Bibliothek realisiert werden kann oder ob eine Verwaltung *aller* Objekte, die der Modulverwaltung unterliegen, im Objektspeicher möglich ist.

Die vorliegende Arbeit gliedert sich wie folgt:

- Im vorliegenden Kapitel motiviere ich im weiteren zunächst, warum es für die Programmierung im Großen sinnvoll, fast unverzichtbar ist, ein Programm in verschiedene Moduln zu gliedern. Daran schließt sich eine kurze Einordnung des Themas in einen größeren Gesamtzusammenhang an.

- In Kapitel 2 werden die Objekte vorgestellt, die Gegenstand der Betrachtung der weiteren Arbeit sein werden. Darüber hinaus werden Belange, die diese Objekte berühren, wie versteckte Bezeichner, Zustände und das Problem des Rautenimports behandelt.

- In Kapitel 3 wird die Umgebung der Modulverwaltung skizziert. Diese ist insofern von Bedeutung, als daß man es hier mit einem persistenten System zu tun hat. Die daraus resultierenden Besonderheiten werden dort erläutert.

- Aufbauend auf den Kapiteln 2 und 3 werden im Kapitel 4 die Aufgaben der Tycoon Modulverwaltung geschildert. Gleichzeitig wird eine Abgrenzung zu Aufgaben gegeben, die derzeit nicht Teil der Modulverwaltung sind, aber durchaus integriert werden könnten.

- Der Zugriff auf die Tycoon Modulverwaltung geschieht bisher mit kaum bzw. unklar dokumentierten Kommandos. Kapitel 5 schließt diese Lücke und legt die Semantik dieser Kommandos offen.

- Kapitel 6 beschäftigt sich mit der Implementierung der Tycoon Modulverwaltung. Es gibt einen Überblick über den internen Aufbau, der anhand zweier ausgewählter Abläufe vertieft wird.

- In Kapitel 7 und 8 wird die praktische Arbeit dokumentiert. Zunächst werden die Arbeiten beschrieben, die direkt am Code der Modulverwaltung vorgenommen wurden. Es handelt sich hierbei sowohl um die Beseitigung aufgetretener Fehler als auch um Verbesserungen im Performanzbereich. Daran schließt sich die Schilderung einer neu entwickelten Komponente zur Systemgenerierung an. Diese wird mit einem vergleichbaren Werkzeug aus der Unix-Welt verglichen und einer abschliessenden Bewertung unterzogen. Zum Schluß dieser Arbeit wird versucht, in einem Ausblick die Richtung für weitere Denkansätze und Entwicklungen aufzuzeigen.

Vom Leser dieser Arbeit wird die Kenntnis der Sprache TL vorausgesetzt.

1.1 Motivation

Das Tycoon System ist entwickelt worden, um besonders die Entwicklung komplexer Anwendungen zu unterstützen. Gerade für die Programmierung dieser großen Systeme, die oft aus hunderttausenden von Zeilen Code bestehen, ist eine besondere Unterstützung nötig, da sie aufgrund ihrer Größe und Komplexität anderenfalls nicht mehr beherrschbar wären. Ein wichtiger Beitrag ist in diesem Zusammenhang die Möglichkeit zur Abstraktion durch Moduln. Dieser Begriff und seine Auswirkungen auf die Programmierung sollen im folgenden erläutert werden.

Nach [Dud88, Wir79] ist ein Modul[1] eine Zusammenfassung von Konstanten, Datentypen, Variablen und Prozeduren zu einer Einheit. Es gliedert sich in ein Definitionsmodul und ein Implementationsmodul. Diese Teile sind oft verschiedene Dateien. Alle Bezeichner, die sich innerhalb des Moduls befinden, sind nur dort sichtbar; umgekehrt sind Bezeichner, die außerhalb definiert sind, nur außerhalb sichtbar. Diese beiden Sichtbarkeitsbereiche werden dadurch, daß einige Bezeichner in der Schnittstelle des Moduls explizit sichtbar gemacht und dann von anderen Moduln und Schnittstellen importiert werden, verbunden. Im Definitionsmodul werden dafür die Bezeichner des Moduls aufgeführt, die für den Klienten des Moduls sichtbar sein sollen, also exportiert werden.[2] Im Implementationsmodul befinden sich die Definitionen für die im Definitionsmodul deklarierten Bezeichner, wie Typen, Variablen etc.

[1] In dieser Arbeit wird der Begriff "Modul" benutzt, der auch in vielen Programmiersprachen, wie z. B. Modula-2, Verwendung findet. In Ada hingegen wird der Begriff des "Paket" (*package*) gebraucht, der semantisch mit dem des Moduls identisch ist.

[2] Die Liste der exportierten Bezeichner wird oft auch mit "Exportschnittstelle" oder einfach "Schnittstelle" bezeichnet. Die Liste der von einem anderen Modul benötigten Bezeichner nennt man "Importschnittstelle".

Hier und evtl. auch im Definitionsteil werden die Bezeichner angegeben, die importiert werden, deren Sichtbarkeitsbereich bis innerhalb des Moduls ausgedehnt wird. Weiterhin werden hier die Prozedurrümpfe der exportierten und lokalen Prozeduren ausprogrammiert. Ein Modul schränkt damit wie eine Prozedur Sichtbarkeitsbereiche ein bzw. legt sie fest [CLR86]. Es bestehen allerdings grundlegende Unterschiede:

- Anders als eine Prozedur, die nur *einen* bestimmten Zweck erfüllt, ist das Modul die Zusammenfassung mehrerer Prozeduren, Datenstrukturen usw. Abhängig von den Objekten und dem Zweck der Zusammenfassung unterscheidet man verschiedene Modularten [CLR86]:

  - Funktionsmodul: Hierunter versteht man ein Modul, das seine Funktionalität mit Hilfe von Prozeduren bzw. Funktionen bereitstellt. Es besitzt keine lokalen Datenobjekte und wird daher auch als gedächtnislos bezeichnet.
  - Dateimodul: Dabei handelt es sich um eine Art von Modul, das Konstanten, Variablen und Datentypen exportiert, z. B. um häufig verwendete Objekte anderen Moduln zur Verfügung zu stellen.
  - Datenmodul: Ein Modul, das ein oder mehrere Datenobjekte implementiert, aber nur die zum Zugriff notwendigen Prozeduren exportiert und die eigentliche Struktur des Objektes verbirgt, bezeichnet man als Datenmodul.
  - Abstrakter Datentyp: Bei dieser Modulart werden ein Datentyp und entsprechende Zugriffsprozeduren exportiert. Damit lassen sich Objekte des Typs erzeugen und bearbeiten. Die Struktur des Typs ist im Modul verborgen.

- In einem Modul lassen sich die Bezeichner selektiv über Import- und Exportlisten sichtbar machen bzw. verstecken. Eine Prozedur kann lokale Bezeichner nicht nach außen sichtbar machen. Globale Programmobjekte können in ihrer Sichtbarkeit nicht weiter eingeschränkt werden. Die einzige Möglichkeit besteht darin, globale Objekte innerhalb einer Prozedur durch lokale Objekte zu überdecken, nicht aber die Globalität aufzuheben.

- Die Lebensdauer lokaler Bezeichner einer Prozedur erstreckt sich nur auf die Zeit der Abarbeitung. Danach verlieren sie ihre Gültigkeit. Bezeichner innerhalb eines Moduls sind über die gesamte Lebensdauer der Modulumgebung definiert und damit statisch[3].

- Der Rumpf eines Moduls wird nur einmal bei der Initialisierung ausgeführt, der Rumpf einer Prozedur bei jedem Aufruf.

Durch das Modulkonzept lassen sich Programme in überschaubare und weitgehend voneinander unabhängige Teile gliedern. Dies hat inbesondere bei der Entwicklung grosser und komplexer Softwaresysteme Vorteile.

[3] In der Programmiersprache C gibt es anders als z. B. in Modula-2 die Möglichkeit, auch innerhalb von Prozeduren statische Variablen anzulegen. Dies geschieht durch die Zuweisung einer Variablen zur Speicherklasse *static*. Die Variable behält dadurch auch nach Beendigung der Prozedur ihren Wert. Allerdings müssen auch alle Bezeichner, die innerhalb eines Moduls deklariert werden und nicht nach außen sichtbar sein sollen, mit *static* aufgeführt werden; sie erlangen anderenfalls programmweite Gültigkeit. Die Speicherklasse *auto* entspricht innerhalb einer Prozedur der aus Pascal oder Modula-2 gewohnten Semantik [KR90].

In [Mey88] findet sich eine auf den Entwurf von Software gerichtete Diskussion der Modularisierung. Hier werden fünf Prinzipien entwickelt, die wiederum aus fünf Kriterien abgeleitet werden und nach denen der Entwurf vorgenommen werden sollte. Zwar ist der Softwareentwurf nicht Thema dieser Arbeit, die Diskussion in [Mey88] bietet aber einige weitere Argumente für die Nützlichkeit der Modularisierung, die im folgenden kurz erläutert werden.

1.1.1 Separate Übersetzung

In der Literatur tauchen zu diesem Thema zwei ähnliche Begriffe auf, die der Klärung bedürfen [Gei83]. Ihre unterschiedliche Bedeutung hat weitreichende Konsequenzen:

Unabhängige Übersetzung: Unter unabhängiger Übersetzung (*independent compilation*) versteht man das Übersetzen einzelner Programmeinheiten, ohne dabei die Schnittstellen anderer Einheiten zu prüfen. Sie wird von vielen Programmiersprachen unterstützt, die keine Modularisierungsmechanismen besitzen.

Separate Übersetzung: Im Gegensatz zur unabhängigen Übersetzung wird bei der separaten Übersetzung (*separate compilation*) jedes Modul auf Konsistenz mit den von ihm benutzten Schnittstellen überprüft.

Beide Techniken gestatten es dem Programmierer, seine Arbeit in kleinere Einheiten zu zerlegen, in Moduln aufzuteilen. Es ist aber offensichtlich, daß die separate Übersetzung, wie sie Modula-2 oder Mesa [LS79] bieten, durch die Erzwingung der Typkonsistenz über Modulgrenzen hinweg größere Vorteile bringt. Fehler, die z. B. auftreten, weil eine Funktion aus einem fremden Modul mit falschen Parametern aufgerufen wird, können schon zur Übersetzungszeit erkannt und beseitigt werden.

Im Vergleich zu monolithischen Systemen, die aus einer großen Programmdatei bestehen oder stets komplett übersetzt werden müssen, bietet separate Übersetzung die Möglichkeit, den Übersetzungsaufwand nach einer Änderung gering zu halten. Es muß meist nur eine kleinere Anzahl der zu einem System gehörenden Einheiten neu übersetzt werden, da andere Moduln, die nicht von der Änderung betroffen sind, unverändert bleiben können.

1.1.2 Kapselung

Man versteht unter Kapselung das Zusammenfassen z. B. eines Datentyps mit den auf ihn anwendbaren Funktionen in der Art, daß nur über diese Funktionen auf den Datentyp zugegriffen werden kann. Programmiersprachen ohne Modulkonzept haben meist keine Mittel, dies darzustellen. Kapselung hängt damit eng mit dem Begriff Abstraktion zusammen. Während Kapselung die "bottom-up"-Sicht repräsentiert, kann man Abstraktion als "top down" -Sichtweise verstehen. Hier wird ausgehend von gewissen Attributen z. B. ein abstrakter Datentyp geformt.

Durch Kapselung ist es also möglich, von unwichtigen Details eines Moduls zu abstrahieren und sich auf das Wesentliche zu konzentrieren (*information hiding*). Dies ist eines der wichtigsten Hilfmittel, um sicher und effizient große Systeme zu entwickeln [Wir79]. Die Schnittstelle eines Moduls kann als ein "black box interface" angesehen werden, das die Operationen usw. spezifiziert. Die Implementation ist vollständig im Implementationsteil des Moduls enthalten und damit für den Benutzer des Moduls unsichtbar.

Auf diese Weise können z. B. maschinenabhängige Teile eines Programms gekapselt werden. Dies erleichtert dann die Portierung auf andere Systeme. Ganz allgemein lassen sich die Auswirkungen von Änderungen besser lokalisieren und damit die Wartbarkeit erhöhen.

1.1.3 Hierarchische Strukturierung

Die Wartungsfreundlichkeit wird auch durch die hierarchische Strukturierung unterstützt. Während bei Programmiersprachen wie C[4] oder Pascal[5] alle Prozeduren in *einem* globalen Sichtbarkeitsbereich definiert sind, bekommt der Entwickler durch Moduln eine weitere Möglichkeit zu strukturieren. Dies hat hauptsächlich zwei Vorteile.

- Es kann verhindert werden, daß verschiedene Prozeduren sich in undisziplinierter Weise gegenseitig benutzen. Stattdessen wird die Benutzt-Relation zwischen den Moduln automatisch ein gerichteter Graph[6].
- Die Wiederverwendung von Moduln bzw. Modulhierarchien wird erleichtert, da die verwendeten Moduln durch die klare Struktur der Abhängigkeitsrelation bekannt sind. Dies kommt vor allem dann zum Tragen, wenn Moduln wiederum in größere Einheiten wie Bibliotheken zusammengefaßt werden können.

1.1.4 Unabhängige Entwicklung

Die Verbindung mehrerer Moduln geschieht ausschließlich über die Schnittstelle. Für die Implementierung eines Modulrumpfes ist es ausreichend, die Schnittstellen aller benutzten Moduln zu kennen. Das Ausprogrammieren verschiedener Moduln kann also völlig unabhängig voneinander geschehen, sind die Schnittstellen einmal festgelegt. Die Entwickler können so simultan an verschiedenen Moduln arbeiten, ohne während des Programmierens weitere Absprachen treffen zu müssen; diese sind mit den Schnittstellen bereits getroffen worden.

Natürlich wird dieser Vorteil aufgehoben, wenn sich Schnittstellen während der Entwicklung häufig ändern. Dies ist, wie [AWT89] zeigt, ein nicht seltener Vorgang.

1.2 Einordnung

Die Modulverwaltung als eine Komponente kann nicht für sich betrachtet werden. Sie ist ein Teil des Softwareentwicklungsprozeßes insofern, als daß ihr eine bestimmte Aufgabe innerhalb dieses Kontextes zukommt. Welche Aufgaben dies sind, soll anhand des Begriffs Konfigurationsmanagement (*configuration management*) und der damit verbundenen Tätigkeiten gezeigt werden. Die Modulverwaltung wird damit in einen größeren Zusammenhang eingeordnet.

Das Konfigurationsmanagement umfaßt eine Reihe verschiedener, teilweise rechnergestützter Vorgänge. Die wichtigsten sind nach [Whi91, Est88, Tic88]:

1. Objektidentifikation (*identification, item identification*)

[4] In C sind alle nicht explizit als lokal definierte Variablen- und Funktionsbezeichner global in allen anderen Übersetzungseinheiten sichtbar [KR90].

[5] Einige Pascal Dialekte wie z. B. UCSD-Pascal [BCHT86] oder Turbo Pascal [Woo89] unterstützen ein Modularisierungskonzept. Die Modularisierungseinheiten werden hier als Units bezeichnet.

[6] In C sind auch zirkuläre Abhängigkeiten zwischen Moduln erlaubt [KR90]. Hingegen ist dies in TL oder in Modula-2 verboten [Mat93, Wir79].

2. Versionierung (*version management, version selection and baselining*)
3. Änderungsverwaltung (*change control, change tracking*)
4. Systemgenerierung (*building the system, software manufacturing, regeneration*)
5. Bibliotheksverwaltung (*library management, rights and protection*)
6. Konkurrierender Zugriff (*managing simultaneous updates*)

Im Rahmen dieser Arbeit sind besonders Punkt eins und vier von Interesse und werden im folgenden dargelegt. An dieser Stelle wird auch darauf hingewiesen, daß wie im Verlauf der Arbeit noch gezeigt wird, die Modulverwaltung auch allgemeiner, also nicht auf das Umfeld von Programmiersystemen beschränkt, aufgefaßt werden kann. Das Haupteinsatzgebiet der existierenden Modulverwaltungen und insbesondere der des Tycoon Systems liegt aber im Programmiersprachenumfeld.

1.2.1 Objektidentifikation

Die Objektidentifikation bildet die Grundlage für alle Tätigkeiten des Konfigurationsmanagements und damit auch für die Systemgenerierung bzw. Modulverwaltung. Sie legt fest, mit welchen Objekten man es zu tun hat und wie diese identifiziert werden. Nur wenn Objekte einzeln benennbar sind, lassen sie sich auch zu größeren Einheiten aggregieren oder andere Beziehungen zwischen ihnen aufbauen.

1.2.2 Systemgenerierung

Die Systemgenerierung erfordert eine Reihe unterschiedlicher Schritte, wie Übersetzen, Vor- und Nachbearbeitung, Binden, Formatieren usw., und die dazu erforderlichen Programme bzw. Prozeduren. Die Modulverwaltung soll diese Transformationen unterstützen und soweit wie möglich automatisieren, geht jedoch noch über den Begriff der Systemgenerierung hinaus. Bei der Generierung muß insbesondere das folgende berücksichtigt werden:

- Die entsprechenden Werkzeuge müssen mit den richtigen Optionen versorgt werden. Z. B. verlangt der GNU[7] C Compiler die Option *-Wall*, damit sämtliche Fehlermeldungen und Warnungen ausgegeben werden.
- Es muß die richtige Version des Werkzeuges verwendet werden. Im Zusammenhang mit dem ersten Punkt erlangt diese Forderung besonders nach einem Update des Werkzeuges Bedeutung. Es muß dann sichergestellt werden, daß stets die alte Version verwendet wird. Es ist natürlich auch möglich, die neue Version des Werkzeuges zu benutzen, wobei dann auf veränderte syntaktische und semantische Eigenschaften des Werkzeuges geachtet werden muß. Es kommt z. B. nach dem Aktualisieren auf eine neue Compiler-Version manchmal vor, daß ohne Änderung des Quelltextes neue Fehlermeldungen beim Übersetzen auftauchen. Dies hat seine Ursache in der geänderten Semantik des Compilers, dem entweder durch Verwendung der alten Version des Compilers oder durch Anpassen des Quelltextes Rechnung getragen werden muß.

[7] GNU: "Gnu's Not Unix" ist der Name für ein Unix-kompatibles Softwaresystem.

- Fehlermeldungen oder Warnungen können während der Transformation auftreten. Diese müssen protokolliert und ggf. behandelt werden.
- Die Reihenfolge der Transformationen ist signifikant.
- Nur die Transformationen sollen ausgeführt werden, die auch wirklich nötig sind.

Ein wichtiger Aspekt ist der Algorithmus, mit dem bestimmt wird, ob eine Komponente transformiert werden muß. Ein einfacher Ansatz ist der des rekursiven Abstiegs in den Abhängigkeitsgraphen und der Benutzung eines Zeitstempels zur Überprüfung der Aktualität einer Komponente, der z. B. von *Make* [Fel79] benutzt wird. Eine Verbesserung hinsichtlich der Anzahl der zu übersetzenden Komponenten und damit der gesamten Rechenzeit stellt die durch Smart Recompilation [TB85] und Smarter Recompilation [SK88] gewählte Herangehensweise dar. Hier wird eine genauere Information über die Abhängigkeiten von Komponenten dazu genutzt, differenziertere Aussagen darüber zu machen, ob eine Komponente transformiert werden muß. Die Tatsache, daß ein oder mehrere Nachfolger einer Komponente im Abhängigkeitsgraphen transformiert wurden, also nicht mehr aktuell waren, führt nur dann zur Transformation der fraglichen Komponente, wenn innerhalb der Komponenten Teile betroffen waren, auf die sich auch die Abhängigkeitsbeziehung begründet. Daneben gibt es weitere Ansätze, die es dem Benutzer überlassen, beliebige Abhängigkeiten zwischen den Komponenten zu definieren und die auszuführenden Transformationen abhängig von weiteren Bedingungen zu machen.

Daß das Feststellen der tatsächlich zu transformierenden Komponenten nicht nur von akademischem Interesse ist, zeigt [AWT89]. Die dortige Untersuchung zweier in Ada geschriebener Systeme (die Ergebnisse sind auf andere Sprachen übertragbar) zeigt, daß zwischen 58% und 62% der getätigten Übersetzungen unnötig waren und durch den Einsatz von Smart Recompilation hätten verhindert werden können. Ein weiteres Ergebnis der Studie unterstreicht die Notwendigkeit effektiver Transformationstechniken. Es hatte sich gezeigt, daß ein Drittel aller Änderungen solche waren, die weitere Transformationen nach sich zogen (Änderung der Spezifikation eines Ada Packets). Im Durchschnitt waren 46% aller Komponenten von solch einer Änderung betroffen.

Auch die Parallelisierung mehrerer unabhängiger Transformationen ist ein zu beachtender Punkt, der nicht nur die gesamte Ausführungszeit günstig beeinflußt, sondern auch die Auslastung des gesamten Systems verbessert.

Mit dem Zusammenbau eines Systems beschäftigt sich auch der im Rahmen dieser Arbeit angefertigte praktische Teil.

Kapitel 2

Objekte der Tycoon Modulverwaltung

In diesem Kapitel werden die in der Sprache TL zur Verfügung stehenden Mittel zur Programmierung im Großen vorgestellt. Gemäß Abschnitt 1.2.1 stellt dies die Grundlage für die weitere Betrachtung dar. Hier werden diejenigen Objekte identifiziert, mit der die Modulverwaltung umgeht und deren Beziehungen sie darstellt.

TL stellt Moduln (*module*), Schnittstellen (*interface*) und Bibliotheken (*library*) bereit, die hier im folgenden als Objekte bezeichnet werden. Sie sind Gegenstand der Tycoon Modulverwaltung. Die Ausführungen stützen sich auf [Mat93] und [MM93].

2.1 Schnittstelle

Schnittstellen enthalten die Signaturen für Funktionen, Variablen, Typen und Typoperatoren. Die Schnittstelle zu einem Modul ist ein benannter Tupel- oder Recordtyp und damit ein Objekt erster Klasse (s. Abbildung 2.1 und 2.2). Bezeichner, die in einer Schnittstelle deklariert sind, werden mittels Punktnotation angesprochen.

Da es beliebig viele Werte eines Typs geben kann, so kann auch eine Schnittstelle von beliebig vielen Moduln implementiert werden. Die in Abbildung 2.1 gezeigte Schnittstelle kann also einmal von dem Modul in Abbildung 2.3 implementiert werden. Es könnte aber auch von einem anderen Modul gleichen Typs implementiert werden, welches z. B. ein Array zur Darstellung des Stapels benutzt.

Ein Unterschied der in Abbildung 2.1 und 2.2 gezeigten Beispiele besteht in der Handhabung zu importierender Bezeichner. Während im ersten Fall alle zu importierenden Moduln und Schnittstellen explizit in der *import*-Phrase angegeben werden müssen, sind im zweiten Beispiel alle im Sichtbarkeitsbereich von *Stack* befindlichen Bezeichner automatisch auch erreichbar.

2.2 Modul

Ein Modul ist die Implementation seiner Schnittstelle. In TL wird ein Modul durch eine Funktion beschrieben, die zu einem Tupelwert evaluiert, der kompatibel zum Typ der entsprechenden Schnittstelle sein muß. Dieser Tupel muß entsprechend der Signaturen der

```
interface Stack
  import
  export
    T <:Oper(E <:Ok) Ok
    new (E <:Ok) :T(E)
    empty (E <:Ok  stack :T(E)) :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 2.1: Schnittstelle für die Implementation eines Stack-Moduls

```
Let Stack = Tuple
  T <:Oper(E <:Ok) Ok
  new (E <:Ok) :T(E)
  empty (E <:Ok  stack :T(E)) :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 2.2: Äquivalente Formulierung der Schnittstelle aus Abbildung 2.1 ohne Benutzung ausgewiesener Modularisierungsmechanismen

Schnittstelle kompatible Bindungen aufweisen. Von dem Modul importierte Bezeichner treten als Parameter der Funktion auf.

Wie in Abbildung 2.4 zu sehen, werden durch eine am Ende der Funktion vorgenommene explizite Bindung des Rückgabewertes drei Dinge erreicht:

- Innerhalb der Funktion (Modulrumpf) können auch lokale, also nicht exportierte, Bindungen vorgenommen werden. Im Beispiel der Abbildung 2.4 wurde hiervon kein Gebrauch gemacht.
- Die Reihenfolge der zu exportierenden Bindungen ist nicht relevant.
- Die Initialisierung des Moduls kann Seiteneffekte aufweisen.

Nach der Definition im Beispiel aus Abbildung 2.4 muß der Parameter der Funktion *stack* durch

```
let stack = stack (list);
```

an den Bezeichner *list* gebunden und die Funktion evaluiert werden. Erst dann läßt sich *stack* so benutzen, als ob es importiert worden wäre.

Intern wird ein Modul repräsentiert durch

```
module stack
  import
    list :List
  export
    Let T(E <:Ok) = list.T(E)
    let new (E <:Ok) :T(E) =
      list.new (:E)
    let empty (E <:Ok  stack :T(E)) :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 2.3: Formulierung eines Moduls mit dem dafür von der Sprache TL bereitgestellten *module*-Konstrukt

```
let stack (list :List) = begin
  Let T(E <:Ok) = list.T(E)
  let new (E <:Ok) :T(E) =
    list.new (:E)
  let empty (E <:Ok  stack :T(E)) :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
end
```

Abbildung 2.4: Zu Abbildung 2.3 äquivalente Darstellung eines Moduls

Bytecode: Dies ist der vom Compiler erzeugte TVM-Code. Dieser kann von der Tycoon Maschine (TVM) ausgeführt werden.

Literale: Hierbei handelt es sich um einen Vektor, der Verweise auf lokale Konstanten enthält, z. B. ein konstanter String, der ausgegeben wird.

Tupelwert: Dies ist der Wert, zu dem das Modul evaluiert wird.

Im Rahmen dieser Arbeit ist hauptsächlich der Tupelwert eines Moduls von Bedeutung. Die weiteren Details dieser Repräsentation sind nicht relevant, da das Modul von der Modulverwaltung stets als Einheit behandelt wird.

```
interface A
export
  T <:Ok
  x :T
end

interface B
import a :A
export
  y :a.T
end

interface C
import a :A
export
  f(z :a.T) :Ok
end

interface D
export
end
```

```
module a :A
export
  Let T = ...
  let x = ...
end

module b :B
import a :A
export
  let y = a.x
end

module c :C
import a :A
export
  let f(z :a.T) = ...
end

module d :D
import b :B  c :C
export
  c.f(b.y)
end
```

Abbildung 2.5: Beispiel zum Problem des Rautenimports (aus [Mat93])

Die Beispiele machen eine weitere bewußte Einschränkung der Orthogonalität der Benennungs-, Bindungs- und Typisierungskonzepte der Sprache TL deutlich. Eine Schachtelung von Moduln und Schnittstellen mit den Konstrukten *module* und *interface* ist nicht möglich. Eine Formulierung ohne diese Phrasen läßt eine Schachtelung jedoch zu.

2.3 Bibliothek

Eine Bibliothek legt die Sichtbarkeitsbereiche für Moduln und Schnittstellen fest. Der Sichtbarkeitsbereich eines Bezeichners ist durch die in der Bibliothek vor ihm aufgeführten Be-

zeichner bestimmt. Dies schließt zyklische Abhängigkeiten zwischen Komponenten aus. Es löst weiterhin das Problem des Rautenimports (s. Abbildung 2.5). In einer Situation, in der zwei unterschiedliche Moduln b und c ein weiteres Modul a importieren und von dem Modul d importiert werden, kann aus der Namensgleichheit des Bezeichners a in b und c gefolgert werden, daß es sich bei der Funktionsapplikation in d um den gleichen Typ $a.T$ handelt [Mat93, Car90].

Bibliotheken können, wie in Abbildung 2.6 dargestellt, geschachtelt werden. Dabei muß unterschieden werden, zwischen Bibliotheken, die andere Bibliotheken innerhalb des *with library*-Konstruktes aufführen. Damit sind alle Bezeichner einer Bibliothek in nachfolgend genannten Bibliotheken bekannt. Darüber hinaus können in einer Bibliothek explizit Bezeichner einer anderen Bibliothek, die in ihrem Sichtbarkeitsbereich liegt, durch *import* eingeführt werden. Dies führt zu zwei Arten von Bibliotheken:

1. Eine Bibliothek bzw. die in ihr deklarierten Moduln und Schnittstellen kann auf die Verwendung von Komponenten aus anderen Bibliotheken angewiesen sein. Eine derartige Bibliothek kann also immer nur als *Subbibliothek* einer anderen Bibliothek auftreten und somit nie Wurzelbibliothek sein (vgl. Abschnitt 3.5). Im Beispiel aus Abbildung 2.6 trifft dies für die Bibliothek *FruitLib* zu.

2. Existieren in einer Bibliothek keine Moduln oder Schnittstellen, die Komponenten aus anderen Bibliotheken importieren, so kann diese Bibliothek als Wurzelbibliothek auftreten (Bibliothek *Root* aus Abbildung 2.6). Außerdem kann diese Bibliothek, ohne daß Namenskonflikte irgendeiner Art entstehen würden, innerhalb eines Bibliothekssystems umgruppiert werden.

Trotz der Möglichkeit zur Schachtelung sind *alle* Bezeichner der Bibliothek *FruitLib* in den Bibliotheken *GreensLib* und *FlowerLib* sichtbar. Es existiert nur *ein* globaler Sichtbarkeitsbereich, in dem sich alle Komponenten befinden und gemäß der oben erwähnten Regel sichtbar sind.

2.4 Versteckte Bezeichner

Mit der *hide*-Klausel kann zusätzlich zum Konzept der Bibliotheken die Sichtbarkeit bzw. Importierbarkeit von Moduln und Schnittstellen eingeschränkt werden. Ein in der *hide*-Klausel aufgeführter Bezeichner ist im Rahmen der entsprechenden Sichtbarkeitsbereiche nur innerhalb der unmittelbar einschließenden Bibliothek importierbar, nicht aber außerhalb.

In der in Abbildung 2.6 aufgeführten Bibliothek ist der Bezeichner *tomato* nur innerhalb der Bibliothek *FruitLib* importierbar, aufgrund der Sichtbarkeitsregeln sogar nur in *peach*. *Tomato* ist außerdem im Modul *apple* und *Peach* sichtbar. Hingegen sind *peach*, *Peach*, *apple* und *Apple* auch im Sichtbarkeitsbereich der Bibliothek *FlowerLib* und können von darin enthaltenen Komponenten importiert werden.

Eine Namensüberdeckung einer versteckten Komponente in einer anderen Bibliothek durch Redefinition des entsprechenden Bezeichners ist nicht möglich, da die Sichtbarkeit von versteckten Bezeichnern nicht eingeschränkt ist. Es könnte also in der Bibliothek *FlowerLib* kein Bezeichner *tomato* oder *Tomato* definiert werden, in diesem Fall käme es zu einer Fehlermeldung. Dies vereinfacht die Reorganisation großer Bibliothekssysteme, da verhindert wird, daß durch Umstrukturierungen neue Namenskonflikte entstehen. *Tomato* und *tomato*

```
library Root
with
  interface
    Field
  module
    field :Field
  library
    FruitLib GreensLib FlowerLib
end

library FruitLib
import field :Field
with
  interface
    Apple
    Tomato
    Peach
  module
    apple :Apple
    tomato :Tomato
    peach :Peach
  hide
    tomato Tomato
end
```

Abbildung 2.6: Geschachtelte Bibliotheken mit einem versteckten Bezeichner

könnten also auch sichtbar in der Bibliothek *GreensLib* deklariert werden anstelle in *FruitLib*, ohne daß ein Namenskonflikt entstünde.

Der Nachteil dieses Ansatzes ist, daß, obwohl Bezeichner in einer Bibliothek versteckt deklariert sind, diese dennoch bekannt sein müssen. Z. B. läßt sich nicht gleichzeitig in der Bibliothek *FruitLib* ein versteckter Bezeichner *tomato* deklarieren und in der Bibliothek *GreensLib* ein Bezeichner gleichen Namens sichtbar oder ebenfalls versteckt. Dies macht die Kombination beliebiger Bibliotheken nur aufgrund der sichtbaren Bezeichner unmöglich.

2.5 Zustand

Übersetzte Komponenten, insbesondere Moduln und Schnittstellen, können sich in verschiedenen Zuständen befinden. Je nach Art der Komponente hat dies unterschiedliche Bedeutungen.

Bibliotheken werden in dieser Diskussion nicht weiter betrachtet. Sie können sich entweder im Objektspeicher befinden und dann Sichtbarkeitsbereiche für andere Komponenten festlegen oder aber nicht geladen sein.

2.5.1 Modul

In Abschnitt 2.2 wurde ausführlich erläutert, daß ein Modul durch eine Funktion repräsentiert wird, die einen Tupelwert zurückgibt. Diese Funktion kann nun in den Objektspeicher *geladen* werden (vgl. Abbildung 2.7). Es ist der erste Zustand, den ein Modul annehmen kann.

Im nächsten Schritt kann die Funktion, parametrisiert mit den Moduln und Schnittstellen, die sie importiert, ausgeführt werden. Das Modul befindet sich dann im Zustand *gebunden*. Dieses Modul bzw. sein Modulwert kann nun von anderen Moduln und Schnittstellen, in deren Importliste es sich befindet, referenziert werden.

Soll der Wert des Moduls für den Benutzer sichtbar sein, so muß er an einen Namen gebunden werden. Dies bedeutet, daß der Wert des Moduls im Toplevel-Vektor eingetragen wird. Das Modul ist damit *definiert*.

Aus der Aufstellung wird deutlich, daß sich die Zustände voraussetzen: Ein Modul kann nur gebunden sein, wenn es auch geladen ist. Ebenso kann nur eine Bindung auf dem Toplevel definiert werden für ein Modul, das bereits gebunden ist. Dies geschieht meist in einem einzigen Schritt durch Import des Moduls.

2.5.2 Schnittstelle

Im Gegensatz zu einem Modul stellt eine Schnittstelle Typinformation dar. Diese kann in den Objektspeicher *geladen* werden.

Ein weiterer Unterschied zum Modul besteht im Zustand *gebunden*. Diesen kann eine Schnittstelle nicht annehmen, da sie keinen Wert, sondern einen Typ repräsentiert. Das Auflösen der externen Verweise innerhalb der Typinformation (de Bruijn-Index) geschieht bei einer Schnittstelle erst, wenn sie auf dem Toplevel definiert wird. Z. B. wird in einem Modul durch

```
import :List
```

nur die Schnittstelle des Moduls *List* bekanntgemacht. Es werden also nur Typinformationen referenziert, die zur Übersetzungszeit relevant sind, nicht aber zur Laufzeit. Die Implementation von dort deklarierten Funktionen etc. kann auf diese Weise also nicht angesprochen werden. Folglich wird die Schnittstelle (ohne Implementation) zum Evaluieren des Modulwertes nicht benötigt.

Wie ein Modul kann eine Schnittstelle bzw. die Typinformation an einen Namen (den Schnittstellennamen) gebunden und auf dem Toplevel definiert werden.

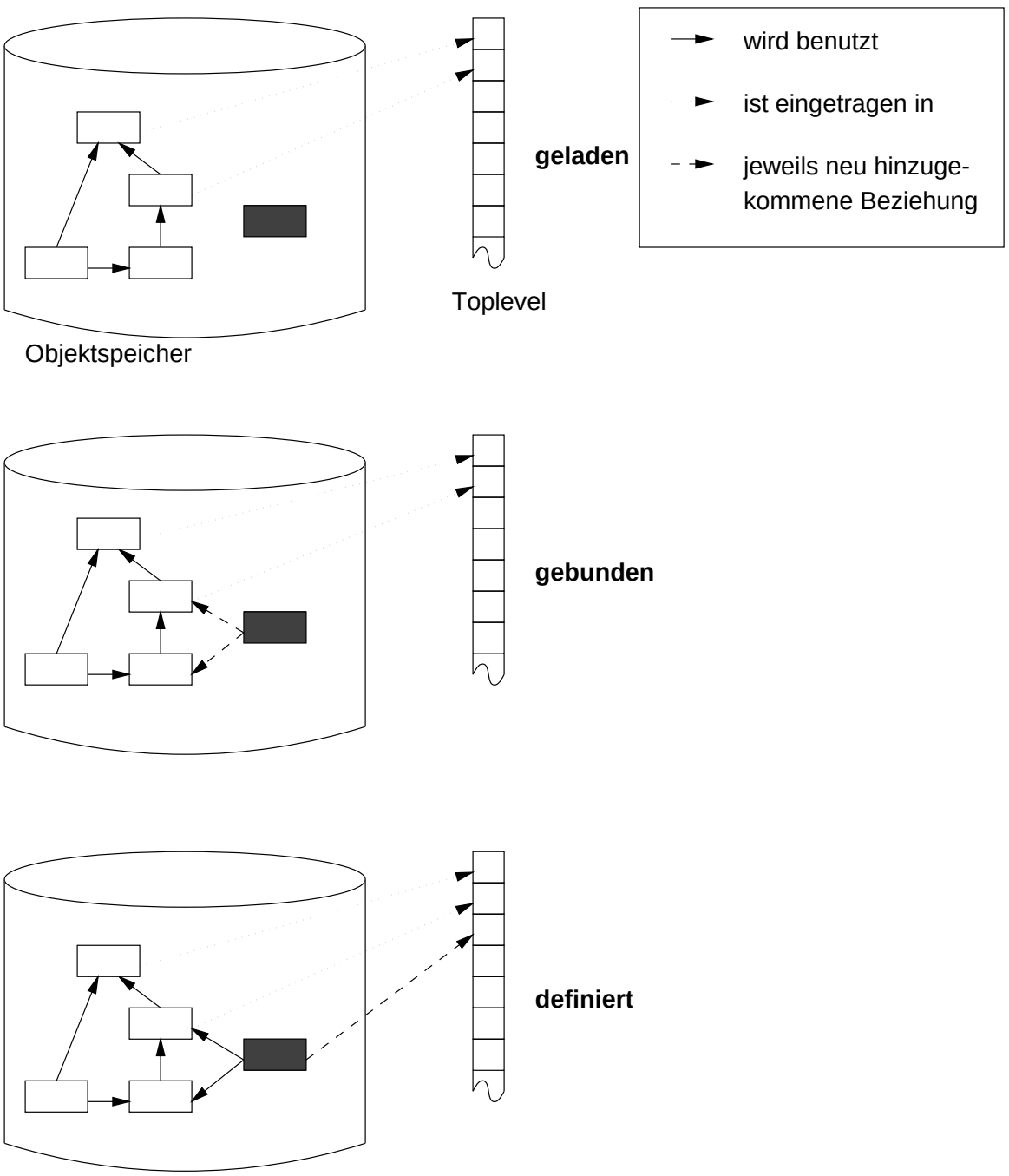

Abbildung 2.7: Zustände, in denen sich ein Modul befinden kann.

Kapitel 3

Umgebung der Tycoon Modulverwaltung

In diesem Kapitel wird geklärt werden, in welcher Beziehung die im vorangegangenen Kapitel erläuterten Objekte der Modulverwaltung zu anderen Komponenten des Tycoon Systems stehen. Hier soll deutlich werden, in welchem Kontext die Modulverwaltung arbeitet. Im Gegensatz zu anderen Systemen handelt es sich bei Tycoon um ein persistentes Objektsystem. Dies impliziert z. B. die Benutzung eines Objektspeichers. Außerdem kann das Tycoon System über den Toplevel auf eine interaktive Weise benutzt werden. Die Aufgaben, die diesen Komponenten im Zusammenhang mit der Modulverwaltung zukommen, werden im folgenden betrachtet. Sie zeigen den Kontext auf, in dem die Modulverwaltung arbeitet. Die hier vorgestellte Umgebung bildet somit die Grundlage für die weitere Analyse.

Zu jedem Modul, jeder Schnittstelle und jeder Bibliothek[1], jedem Objekt der Modulverwaltung also, existieren unterschiedliche Repräsentationen, wie z. B.:

- der Quelltext
- der übersetzte Quelltext
- der abstrakte TL Syntaxbaum
- der abstrakte TML Syntaxbaum
- der Bytecode
- evtl. C-Code[2]

Diese Repräsentationen sind insofern interessant, als daß sie an unterschiedlichen Lokationen abgelegt werden und es Werkzeuge geben muß, die Abbildungen zwischen den Repräsentationen vornehmen können. In dieser Arbeit werden nur die Darstellung als Quelltext und als übersetzter Quelltext näher betrachtet. Abstrakter TL und TML Syntaxbaum und

[1] Für eine Bibliothek existieren nicht alle der aufgeführten Repräsentationen, sondern von der vorliegenden Liste nur die ersten beiden.

[2] Der Tycoon Compiler erzeugt aus dem TL -Code in mehreren Schritten Bytecode, der anschließend von der Tycoon Maschine interpretiert wird. Normalerweise erfolgt keine Übersetzung in C-Code und danach von einem C-Compiler in Maschinencode. Es gibt jedoch durchaus die Möglichkeit, auch C-Code zu generieren und damit Verbesserungen im Laufzeitverhalten zu erreichen.

der Bytecode werden vom Compiler erzeugt. Der Bytecode schließlich wird von der Tycoon Maschine ausgeführt.

3.1 Objektspeicher

Im Objektspeicher werden beliebig strukturierte TL -Objekte abgelegt, nachdem sie während einer Tycoon-Sitzung erzeugt wurden. Sie können explizit persistent gemacht werden. Eine Freispeicherverwaltung (*garbage collection*) sorgt dafür, daß nicht mehr erreichbare Objekte automatisch aus dem Objektspeicher entfernt werden.

Dauerhaft werden hier nur übersetzte Komponenten abgelegt. Die Speicherung des Quelltextes erfolgt nicht im Objektspeicher, obwohl dies möglich wäre. Der Grund hierfür liegt in der Tatsache, daß die zur Bearbeitung des Quelltextes im Objektspeicher nötigen Werkzeuge zum Editieren, Drucken, Formatieren aber z. B. auch zum Suchen in mehreren Quelltexten zugleich etc. nicht oder nicht in genügender Qualität vorhanden sind. Allgemein verfügbare Werkzeuge, die die entsprechenden Aufgaben erfüllen, sind zum Lesen und Schreiben von Objekten stets an das Dateisystem gebunden und können nicht auf den Objektspeicher zugreifen.

In der Praxis widerspricht ein weiterer Grund dem Ablegen von Quelltexten im Objektspeicher. Zum einen laufen die gegenwärtigen Implementierungen noch nicht ausreichend stabil, um tatsächlich eine Datensicherheit und damit Persistenz garantieren zu können. Häufige Abstürze würden dann jeweils zum Verlust der gesamten Quelltexte führen. Zum anderen fehlt die Möglichkeit, ein Backup eines Objektspeichers anzulegen. Dies könnte zu einem gewissen Teil den ersten Nachteil wieder aufheben. Bislang ist eine Datensicherung nur dann möglich, wenn der Objektspeicher an das Dateisystem gebunden ist. In diesem Fall müssen jedoch externe Werkzeuge benutzt werden; für Implementierungen, die nicht an das Dateisystem gebunden sind, gibt es keine Backup-Möglichkeiten.

3.2 Externer Speicher

Neben dem Objektspeicher wird als persistenter Speicher auch das auf der Rechnerarchitektur verfügbare Dateisystem genutzt. Der Objektspeicher ist je nach Implementation[3] ebenfalls an das Dateisystem gebunden.

Das Tycoon System ist auf vielen Architekturen verfügbar. Infolgedessen muß es mit den verschiedensten Dateisystemen zusammenarbeiten und dennoch zum Rest des Systems eine definierte Schnittstelle für den Zugriff darauf bereitstellen, die in allen Systemen Gültigkeit hat. Besonders die Beschränkung in der Länge von Dateinamen oder das Trennzeichen zwischen Komponenten eines Pfadnamens können hier Probleme darstellen.

Es wurde bereits erläutert, wie Moduln, Schnittstellen und Bibliotheken in TL und im Objektspeicher repräsentiert werden. Externer Speicher dient dazu, sowohl die Quelltextrepräsentation als auch übersetzte Objekte abzulegen. Die entsprechende Verzeichnisstruktur wird durch die Umgebungsvariablen *librarySourceDir* und *libraryObjectDir* festgelegt (s. a. Tabelle 3.1).

[3]Die derzeit am häufigsten genutzten Objektspeicherimplementierungen *tymem* und *tysin* nutzen entweder Dateien oder den Hauptspeicher des Rechners, der bei *do saveSystem* dann in einer Datei gespeichert wird.

Art der Datei	Dateiname	Verzeichnis
Quelltext einer Bibliothek *lib*	*lib*.tl	*librarySourceDir/lib*
Quelltext eines Moduls *mod* aus *lib*	*mod*.tm	*librarySourceDir/lib*
Quelltext einer Schnittstelle *itf* aus *lib*	*itf*.ti	*librarySourceDir/lib*
Übersetzte Bibliothek *lib*	*lib*.tl.x	*libraryObjectDir/lib*
Übersetztes Modul *mod* aus *lib*	*mod*.tm.x	*libraryObjectDir/lib*
Übersetzte Schnittstelle *itf* aus *lib*	*itf*.ti.x	*libraryObjectDir/lib*

Tabelle 3.1: Dateiarten mit ihrer Bedeutung und Lokation

3.3 Fingerprint

In nahezu jedem Dateisystem werden die Dateien mit einem Zeitstempel abgelegt, der die Zeit der letzten Änderung[4] festhält. Diese Zeitmarke ist wichtig, um festzustellen, ob die übersetzte Datei und der Quelltext noch konsistent sind. Weist der Zeitstempel des Quelltextes ein jüngeres Datum auf als die Zeitmarke des entsprechenden übersetzten Objektes, so kann man annehmen, daß der Quelltext nach der letzten Übersetzung geändert wurde. Es kann allerdings in diesem Fall nicht notwendigerweise davon ausgegangen werden, daß beide Objekte inkonsistent sind. Wurde z. B. nur ein Kommentar geändert oder hinzugefügt, entspricht der übersetzte Text immer noch seinem Quelltext. Um diese Situationen zu erkennen und daraus resultierende Übersetzungsvorgänge zu vermeiden, enthalten die übersetzten Einheiten des Tycoon Systems zusätzlich eine spezielle Prüfsumme (*Fingerprint*) des Quelltextes. Soll dann überprüft werden, ob ein Quelltext der Übersetzung bedarf, wird neben der üblichen Prüfung der Zeitstempel sein Fingerprint berechnet. Sind dieser und der bei der übersetzten Einheit gespeicherte gleich, bedeutet das, daß eine erneute Übersetzung überflüssig ist. Der Quelltext hat sich nicht signifikant geändert.

Zur Berechnung dieser Prüfsumme werden nur für die Sprache relevante Teile des Quelltextes herangezogen. Alle anderen Elemente eines Quelltextes, wie z. B. Kommentare oder Leerzeilen, werden für die Berechnung nicht betrachtet.

Durch den Fingerprint können zwar unnötige Übersetzungsvorgänge vermieden werden, es steht ihm allerdings ein Nachteil gegenüber. Der Compiler verweist bei Fehlermeldungen auf das Auftreten der relevanten Information mittels Zeilennummer und Spaltenangabe. Diese Information ist jedoch ungültig, wenn der Quelltext sich geändert, der übersetzte Text aber unverändert geblieben ist.

3.4 Toplevel

Der Zugang zum Tycoon System erfolgt für den Benutzer üblicherweise über den interaktiven Toplevel. Vergleichbar mit der Eingabezeile eines Kommandozeileninterpreters in UNIX kann

[4] In vielen Dateisystemen, die mit Unix Anwendung finden, werden zu jeder Datei drei Zeitmarken festgehalten:

- die Zeit der letzten Veränderung
- die Zeit der letzten Statusänderung
- die Zeit des letzten Zugriffs

In diesem Zusammenhang ist die erste der genannten Zeiten die ausschlaggebende.

der Benutzer Befehle an das Tycoon System absetzen und erhält Ergebnisse, Warnungen, Fehlermeldungen etc. zurück. Der Toplevel ist die Benutzerschnittstelle[5] des Tycoon Systems.

Alle Tätigkeiten, die der Benutzer ausführen möchte, startet er durch Kommandos auf dem Toplevel, die dann in einer zyklisch durchlaufenen Schleife abgearbeitet werden (*top level loop*). Zu diesen Kommandos gehört z. B. das Setzen von Systemparametern aber auch die für die Modulverwaltung wichtigen Tätigkeiten des Übersetzen, Binden und Ausführen. Bei der standardmäßig verwendeten Grammatik werden alle diese Kommandos durch das Wort *do* eingeleitet, gefolgt von dem entsprechenden Befehl mit seinen Argumenten. Eine genaue Beschreibung der für die Modulverwaltung relevanten Toplevel-Kommandos wird in Kapitel 5 gegeben.[6]

Neben der Möglichkeit direkt Befehle abzusetzen, können auch Skripten benutzt werden. Diese werden dann durch Eingabe von

```
do "Pfadname"
```

abgearbeitet.

Außer den erwähnten Kommandos kann der Benutzer auf dem Toplevel auch beliebig strukturierte TL -Terme eingeben. Insbesondere hat er die Möglichkeit, Werte an Namen zu binden. Diese Namen sind dann auf dem Toplevel sichtbar, man spricht auch von Toplevel-Bindungen. Auch Moduln und Schnittstellen können auf diese Weise an Namen, den Modul- bzw. Schnittstellennamen, gebunden werden und sind dann auf dem Toplevel sichtbar. Im Unterschied zu anderen Bindungen, werden diese meist durch das spezielle Konstrukt *import* angelegt (s. Abschnitt 5.1). Auf diese Weise definierte Moduln und Schnittstellen befinden sich nach Abschnitt 2.5 im Zustand *definiert*. An dieser Stelle wird auf die etwas verwirrende Begriffsbildung hingewiesen. TL -Objekte können an Namen *gebunden* werden und sind dann *definiert*. Im Zusammenhang mit Moduln versteht man unter *Binden* (vom englischen *link*) den Vorgang, Referenzen auf externe Objekte aufzulösen.

Es ist auch denkbar, Moduln und Schnittstellen auf dem Toplevel direkt einzugeben. Dabei ist jedoch zu bedenken, daß es dann keine korrespondierende Dateien gibt. Auch ist aus den in Abschnitt 3.1 genannten Gründen eine Bearbeitung schwierig.

3.5 Wurzelbibliothek

Innerhalb einer interaktiven Sitzung, in der der Benutzer Phrasen auf dem Toplevel eingeben kann, gibt es stets eine ausgezeichnete Bibliothek, die die Sichtbarkeitsbereiche aller importierten Bezeichner festlegt, die *Wurzelbibliothek*. Sie weist die in Abschnitt 2.3 genannten Merkmale auf.

[5] In [Mat93] wird auch auf eine fensterorientierte Benutzerschnittstelle hingewiesen. Abhängig vom verwendeten Betriebsystem kann die Benutzerschnittstelle unterschiedliche Gestalt annehmen. Die interaktive Arbeitsweise am Toplevel ist aber auf jeden Fall möglich.

[6] Eine vollständige Liste aller Toplevel-Kommandos mit einer kurzen Beschreibung erhält man durch Eingabe von *do help;*.

Kapitel 4

Aufgaben der Tycoon Modulverwaltung

Nachdem nun die Objekte der Modulverwaltung und das Umfeld, in dem diese sich bewegen, erläutert wurde, arbeite ich in diesem Kapitel die allgemeinen Aufgaben der Modulverwaltung heraus. Hier soll der Ausgangspunkt für die Entwicklung eines neuen Modulmanagers liegen. Dazu wird gezeigt werden, welche dieser Aufgaben Tycoon-spezifisch und welche von genereller Natur sind. Die generellen Aufgaben bilden dann die Grundlage für einen von speziellen Tycoon Spezifika unabhängigen Modulmanager. Dieser soll die geschilderten Aufgaben in ihrer allgemeinsten Form lösen und durch entsprechende Parametrisierung auch auf das Tycoon System anwendbar sein.

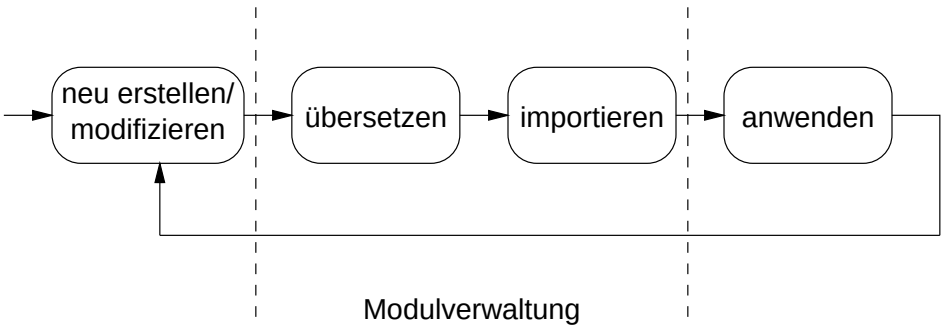

Abbildung 4.1: Typischer Entwicklungszyklus[1] eines Tycoon Quelltext-Objektes

Die Entwicklung einer Tycoon-Komponente ist üblicherweise ein Iterationsprozeß (s. Abbildung 4.1), der die Schritte *Modifizieren*, *Übersetzen*, *Importieren* und *Anwenden* beinhaltet. Sie laufen in einem Zyklus ab, der im Prinzip nie verlassen wird ("Ein Programm ist nie fertig"). Aus dem Blickwinkel der Modulverwaltung sind *Übersetzen* und *Importieren* relevant, da genau diese Aufgaben dort erledigt werden sollen. Sie werden zusammen mt dazu notwendigen bzw. eng zusammenhängenden Tätigkeiten im weiteren beschrieben.

[1]Die Abbildung zeigt einen Zyklus, der nach dem Anwenden wieder beim Modifizieren beginnt. Die Darstellung ließe sich noch beliebig verfeinern. Eine sinnvolle Erweiterung wäre z. B. vor dem Anwenden noch die Tätigkeit Testen, gefolgt von einer Wahlmöglichkeit, ob der Test erfolgreich war, zu modellieren. Dies würde den häufig vorkommenden Fall stärker berücksichtigen, daß ein Entwickler auf dem Toplevel ein geändertes Modul übersetzt, anschließend sofort importiert, dieses daraufhin ausprobiert und weitere Änderungen durchführt. Für dieses Testen muß man also nicht unbedingt den Begriff Anwenden verwenden.

4.1 Übersetzen

Der Übersetzungs- bzw. Systemgenerierungsvorgang selbst, bestehend aus Typüberprüfung, Codegenerierung etc., ist nicht Teil der Modulverwaltung, wird jedoch von ihr gesteuert. Dazu ist es nötig, für jede zu übersetzende Komponente, alle importierten Moduln und Schnittstellen zu ermitteln. Diese werden dann einzeln dem Compiler übergeben, der sie übersetzt.

Diese Aufgabe läßt sich somit als parametrisierter Vorgang auffassen. Der eigentliche Vorgang besteht darin, nach einer vorgegebenen Methode die Importe bzw. Abhängigkeiten zu ermitteln. Daran anschließend werden die entsprechenden Objekte weiter verarbeitet. Die Parameter stellen dabei die Methoden zur Abhängigkeitsermittlung und zum Weiterverarbeiten dar.

4.2 Binden und Definieren

Eine weitere Aufgabe besteht darin, die in den Abschnitten 2.1 und 2.2 beschriebenen Bindungen vorzunehmen. Hierbei handelt es sich zum Teil um eine Tycoon-spezifische Aufgabe. Objekte, die zuvor importiert wurden, sollen nun dem Benutzer sichtbar gemacht und Referenzen zu externen Objekten aufgelöst werden. Hierzu müssen sie in die entsprechenden Strukturen des Tycoon-Systems, insbesondere in den Toplevel-Vektor eingetragen werden. Genauere und ausführlichere Erläuterungen dazu folgen in den Abschnitten 5.1 und 6.2.1.

Das Auflösen externer Referenzen ist ein Vorgang, der auch in anderen Programmiersystemen Anwendung findet, z. B. durch das Bindeprogramm (*linker*) in der UNIX-Umgebung. Dieser ist in der Lage, Objektdateien, die von beliebigen Übersetzern erzeugt wurden, zu einer ausführbaren Einheit zu binden. Insofern handelt es sich beim Bindevorgang um eine generische Aufgabe. Das Bindeprogramm ist jedoch auf ein bestimmtes Format der Objektdateien angewiesen, um die Referenzen zu lokalisieren. Dieses Format ist also der Parameter des Bindevorgangs.

Die Besonderheiten bezüglich des Tycoon Systems ergeben sich nun z. B. aus der Tatsache, daß Objekte explizit auf dem Toplevel bekanntgemacht werden müssen. Vergleichbares muß das Bindeprogramm unter UNIX nicht berücksichtigen, da alle Objekte als Dateien vorliegen und somit dem Benutzer bekannt sind. Weitere in diesem Zusammenhang zu beachtende Punkte, wie z. B. der des Rautenimport [Car90], machen die Spezialisierung auf das Tycoon System deutlich (s. a. Abschnitt 2.3). Es bleibt festzuhalten, daß der Bindevorgang an sich, also nur die Auflösung der externen Referenzen, ein generischer Vorgang ist. Darüber hinausgehende Tätigkeiten sind sehr systemabhängig.

4.3 Sekundärspeicherzugriff

Die Modulverwaltung muß mit mehreren persistenten Speichern arbeiten (s. a. Abbildung 4.2). Zum einen verwaltet sie Objekte, die sich im Objektspeicher befinden, zum anderen greift sie auf Objekte zu, die im Dateisystem liegen. Die Verbindung zwischen den in Kapitel 3 vorgestellten Komponenten Objektspeicher und Sekundärspeicher wird bereits durch die generische Dienste *intern* und *extern* des Moduls *dynamic* realisiert. In der Modulverwaltung werden diese benutzt, indem z. B. das Layout für die zu speichernden Objekte festgelegt wird. Sie bilden damit das Grundgerüst, auf dem die zuvor geschilderten Aufgaben aufsetzen.

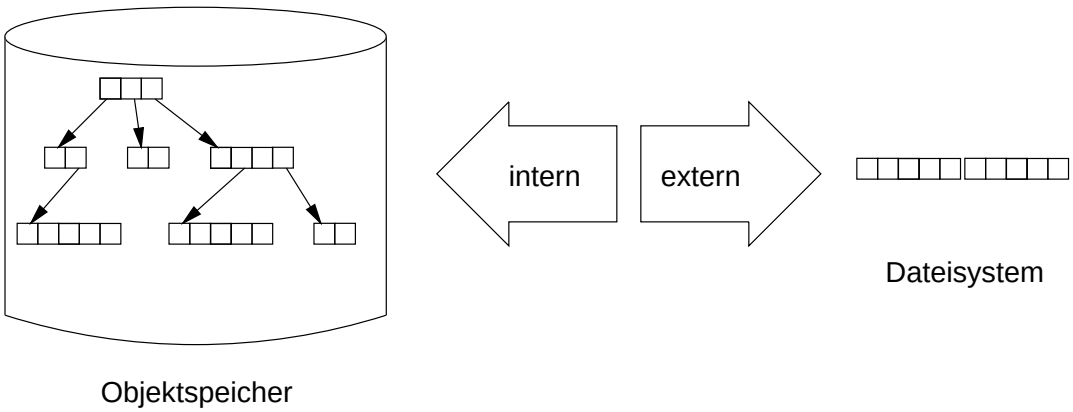

Abbildung 4.2: Der Austausch zwischen Objektspeicher und Dateisystem erfolgt über die Funktionen *dynamic.intern* und *dynamic.extern*.

4.4 Versionierung und Konfigurationsmanagement

Versionsverwaltung und darüber hinaus gehende Aufgabe des Konfigurationsmanagements, die für die Entwicklung großer Softwaresysteme unverzichtbar sind, gehören im Moment nicht zu den Aufgaben der Modulverwaltung. In der am Arbeitsbereich DBIS verwendeten Umgebung werden die Versionsverwaltung und die Verwaltung unterschiedlicher Arbeitsbereiche (*workspace*) der Entwickler durch externe Werkzeuge realisiert und bezieht sich ausschließlich auf Sekundärspeicherobjekte.

4.5 Zusammenfassung

Wie die vorstehenden Ausführungen zeigen, gliedern sich die Aufgaben der Tycoon Modulverwaltung in einen Teil, der speziell auf die Belange von Tycoon und die Sprache TL abgestimmt ist, und in einen Teil, der allgemeine Aufgaben betrifft. Ziel sollte es sein, diese beiden Teile voneinander zu trennen und die allgemeinen Aufgaben in generischer Weise zu lösen. Man erhielte dadurch ein Tycoon System mit klaren Schnittstellen zu anderen Diensten und gleichzeitig wiederverwendbare Softwarekomponenten.

Das in praktischer Arbeit von mir entwickelte Modul *Make* verfolgt diesen Weg in bezug auf die in Abschnitt 4.1 dargelegte Aufgabe. Es wird in Kapitel 8 vorgestellt. Dabei wird besonders versucht, das generische Potential dieser Aufgabe auszuschöpfen.

Kapitel 5

Semantik der Toplevel-Kommandos

Ein Teil der im vorangegangenen Kapitel beschriebenen Aufgaben einer Modulverwaltung werden im Tycoon System durch Toplevel-Kommandos implementiert (s. Abschnitt 3.4). Da sich die verfügbare Dokumentation hauptsächlich auf Anwendung, nicht aber auf die Semantik dieser Kommandos konzentriert, soll dies an dieser Stelle nachgeholt werden.

Der Beschreibung der Semantik der Toplevel-Kommandos, die der *Modulverwaltung* zugerechnet werden können, wurden im wesentlichen vier Fragen zugrundegelegt.

1. Welche Objekte sind betroffen? Werden z. B. Bibliotheken, Moduln oder Schnittstellen durch das Kommando berührt, und in welchen Beziehungen stehen diese zueinander?

2. Was geschieht mit den Objekten? Konkret also, welche Operationen werden auf diesen oder mit diesen durchgeführt? Entstehen weitere Objekte, wie werden sie benannt und wo werden sie abgelegt? In welcher Reihenfolge werden die Objekte bearbeitet?

3. Unter welchen Voraussetzungen werden Aktionen durchgeführt? Die Übersetzung eines Moduls muß z. B. nur in bestimmten Fällen angestoßen werden. Welche Bedingungen müssen erfüllt sein und, welche Daten werden zur Überprüfung dieser Bedingungen herangezogen?

4. Was geschieht im Falle eines Fehlers? Werden einige Aktionen dennoch durchgeführt? Wird das Kommando nach dem ersten Fehler abgebrochen oder erfolgt eine weitere Bearbeitung?

Vor der Beschreibung der einzelnen Kommandos sei noch einmal auf die Begriffsklärung von Abschnitt 2.5 verwiesen. Eine Komponente kann sich, je nach Kommando, das auf sie angewendet wurde bzw. noch nicht angewendet wurde, in verschiedenen Zuständen befinden.

5.1 import *components*

Mit *import components* können ein oder mehrere Moduln oder Schnittstellen in den Objektspeicher geladen und entsprechend Toplevel-Bindungen definiert werden. Bei *components* handelt es sich um eine Liste, die sowohl Modul- als auch Schnittstellen-Bezeichner enthalten kann. Die Komponenten werden der Reihe nach von links nach rechts abgearbeitet. Voraussetzung für das erfolgreiche Importieren ist das Vorhandensein des ausführbaren Codes

aller auch transitiv erreichbaren Komponenten. Außerdem müssen sich alle Komponenten im Sichtbarkeitsbereich der aktuellen Wurzelbibliothek befinden.

Handelt es sich bei einer Komponente um ein Modul, so wird zuerst die zugehörige Schnittstelle importiert und dessen ausführbarer Code geladen (im folgenden wird nicht explizit auf den ausführbaren Code Bezug genommen, dieser ist aber immer gemeint, wenn es darum geht, daß Komponenten geladen bzw. gebunden werden etc.). Folgende Eingaben führen daher zu denselben Bindungen in exakt derselben Ausführungsreihenfolge:

```
import ascii;
import :Ascii ascii;
import ascii :Ascii;
```

Die Schnittstelle wird im ersten Fall aus der aktuellen Wurzelbibliothek bestimmt. Auch im dritten Fall wird zunächst die Schnittstelle *Ascii* importiert; das explizit genannte *:Ascii* hat also keine Wirkung, da es sich zu diesem Zeitpunkt bereits im Objektspeicher befindet. Nur der zweite Fall entspricht also genau der Reihenfolge, in der auch die Modulverwaltung des Tycoon-Compiler die Bearbeitung vornimmt.

Enthalten ein Modul oder eine Schnittstelle in der *import*-Klausel weitere Moduln oder Schnittstellen, so werden zunächst diese geladen und Moduln bzw. deren evaluierter Tupelwert gebunden. Dieser Vorgang geschieht in einer rekursiven Weise, so daß ein Modul erst dann gebunden wird, wenn bereits alle Komponenten, von denen es abhängig ist, geladen und evtl. gebunden sind. Da ein Modul von seiner Schnittstelle implizit abhängig ist, müssen auch, wie oben beschrieben, zunächst die Schnittstelle und dann das Modul importiert werden.

Ist der Lade- und Bindevorgang für alle betroffenen Komponenten abgeschlossen, werden Toplevel-Bindungen definiert. Dies geschieht für alle explizit in *components* angegebenen Moduln und (explizit oder implizit angegebenen) Schnittstellen und für alle Komponenten, die durch diese Schnittstellen sichtbar geworden sind.

Befindet sich eine Komponente, die durch den oben beschriebenen Vorgang direkt oder indirekt betroffen ist, bereits im Objektspeicher, so wird diese nicht ein weiteres Mal geladen oder gebunden. Auch eine Überprüfung, ob der übersetzte Code der Datei aktueller als der im Objektspeicher ist, findet nicht statt, sondern muß explizit durchgeführt werden, z. B. durch das neu eingeführte Kommando *do snycComponent* (s. Abschnitt 7.5).

5.2 do makeComponent *component*

Dieses Kommando bewirkt die Übersetzung des Quellcodes von *component* und aller Schnittstellen, die transitiv von *component* erreichbar sind. Eine Übersetzung findet dabei nur statt, wenn mindestens eine der folgenden Bedingungen erfüllt ist:

- Es existiert bisher noch kein übersetzter Code.
- Der Quellcode wurde seit der letzten Übersetzung signifikant geändert, d. h. der Fingerprint des Quellcodes und des übersetzten Codes sind unterschiedlich.
- Eine oder mehrere von *component* importierte Schnittstellen wurden seit der letzten Übersetzung ebenfalls übersetzt.

Die Reihenfolge der Übersetzung wird dadurch festgelegt, daß zuerst versucht wird, alle importierten Komponenten zu übersetzen.

Weiterhin muß sich *component* im Sichtbarkeitsbereich der Wurzelbibliothek befinden. Unter der Annahme, daß sich *component* in der Bibliothek *library* befindet, erwartet der Compiler den Quellcode im Verzeichnis *librarySourceDir/library* und legt den übersetzten Code in *libraryObjectDir/library* unter dem Namen *component.tm.x* im Falle eines Moduls und *component.ti.x* im Falle einer Schnittstelle ab. *librarySourceDir* und *libraryObjectDir* sind dabei Variable, die mit *do set* entsprechend gesetzt werden können.

Ist bei der Übersetzung einer der betroffenen Komponenten ein Fehler aufgetreten, so wird der gesamte Übersetzungsvorgang abgebrochen und eine entsprechende Meldung ausgegeben.

5.3 do make *component*

Im Gegensatz zu *do makeComponent* betrifft dieses Kommando sowohl Schnittstellen als auch Moduln. Es werden also zusätzlich alle von *component* transitiv importierten Moduln geladen und ggf. übersetzt. Anschließend ist *component* importierbar, d. h. in diesem Fall wird die Komponente gebunden und auf dem Toplevel definiert, da *component* selbst und alle transitiv erreichbaren Komponenten bereits beim Übersetzungsvorgang geladen wurden.

5.4 do makeLibraries *component*

Eine Bibliothek legt für die enthaltenen Moduln und Schnittstellen den Sichtbarkeitsbereich fest. Eine entsprechende Beschreibung wird vom Compiler durch *do makeLibraries* erzeugt. Die Übersetzung einer Bibliothek erfolgt nur, wenn eine oder mehrere der folgenden Bedingungen erfüllt sind:

- Bisher existiert keine übersetzte Bibliothek. Nur Quelltext wurde bisher erstellt.
- Es existiert keine Quelldatei der Bibliothek. Dieser Fall tritt dann auf, wenn die Bibliothek direkt auf dem Toplevel eingegeben wird ohne, daß eine Datei existiert.
- Der Fingerprint der Quelldatei und der Objektdatei sind unterschiedlich.
- Eine in der Bibliothek *lib1* enthaltene Bibliothek *lib2* wurde nach der letzten Übersetzung von *lib1* neu übersetzt.

Nach erfolgreicher Übersetzung einer Bibliothek wird diese zur aktuellen Wurzelbibliothek. Alle Bezeichner, die bereits gebunden wurden, bleiben dabei erhalten, auch wenn sie nicht im Sichtbarkeitsbereich der neuen Bibliothek liegen. Enthält eine Bibliothek weitere Bibliotheken, so diese werden zunächst übersetzt.

5.5 do updateRootLibrary *component*

Dieses Kommando lädt die übersetzte Beschreibung der Bibliothek *component* und ersetzt damit die der Wurzelbibliothek. Es setzt voraus, daß sich die übersetzte Bibliothek im Verzeichnis *libraryObjectDir/component* unter dem Namen *component.tl.x* befindet. Ist dies nicht der Fall, so wird die Ausführung mit einem entsprechenden Fehlerhinweis abgebrochen.

5.6 do loadComponent *component*

Mit diesem Kommando ist es möglich, bereits im Objektspeicher befindliche Komponenten zu aktualisieren oder neue Komponenten zu laden, soweit sich diese im Sichtbarkeitsbereich der Wurzelbibliothek befinden. Außerdem muß die Version der Komponente im Dateisystem aktueller als die des Objektspeichers sein. Abhängig von *component* werden folgende Fälle unterschieden:

- Bei einem Modul erfolgt keine erneute Evaluation des Modulwertes. Die alte Version bleibt nach wie vor gebunden.
- Im Falle einer Schnittstelle bleibt die Typinformation der alten Schnittstelle erhalten.
- Ist das Argument von *do loadComponent* eine Bibliothek, so werden alle transitiv erreichbaren Komponenten gelesen.

Konnte eine Komponente nicht geladen werden, so wird ein entsprechender Hinweis ausgegeben und evtl. mit dem Laden anderer Komponenten fortgefahren. Die alte Version bleibt unverändert im Objektspeicher.

5.7 do link *component*

Die Semantik dieses Kommandos ist dem von *import* sehr ähnlich: Die Komponente *component* wird in den Objektspeicher gelesen, und im Falle eines Moduls wird der Modulrumpf evaluiert. Eine Komponente wird erst dann gebunden, wenn alle transitiv erreichbaren Komponenten gebunden sind. Es bestehen die folgenden Unterschiede zu *import*:

- Argument von *do link* kann auch eine Bibliothek sein. Es werden dann alle auch transitiv in dieser Bibliothek und in eingeschlossenen Bibliotheken enthaltenen Moduln und Schnittstellen gebunden.
- Für kein Modul und keine Schnittstelle wird eine Bindung auf dem Toplevel angelegt.

5.8 do unlink *component*

Es werden abhängig von *component* die folgenden Fälle unterschieden:

Modul: Die Komponente wird als *nicht gebunden* markiert. Folglich kann diese Komponente ein weiteres Mal gebunden werden. Unabhängig vom Binde-Status ist die Komponente weiterhin auf dem Toplevel definiert.

Bibliothek: Alle von der Bibliothek aus transitiv erreichbaren Moduln werden als *nicht gebunden* markiert.

Im Zusammenspiel mit *do link* kann so die erneute Evaluierung des Modulrumpfes erreicht werden. Dies ist insbesondere dann sinnvoll, wenn die Modulinitialisierung Seiteneffekte aufweist.

5.9 Bewertung

Die häufig benutzten Kommandos *import*, *do make*, *do makeComponent* und *do makeLibraries* sind in ihrer Semantik so gefaßt, daß sie vom Benutzer intuitiv verwendet werden können. Dies trifft allerdings nicht auf doe Kommandos *do loadComponent*, *do link* und *do unlink* zu. Ihr praktischer Nutzen ist unklar. So ist es nicht sinnvoll, Komponenten explizit zu laden. Dies wird auch automatisch von *import* erledigt. Der Nutzen der Kommandos *do link* und *do unlink* ist insofern eingeschränkt, als daß das damit intendierte Neuberechnen des Modulwertes nicht klar erkennbar ist. Es wäre sinnvoller, auch um mögliche Inkonsistenzen zu verhindern, diese Tätigkeit in einem einzigen Kommando zusammenzufassen. Abbildung 5.1 macht den Zusammenhang zwischen Toplevel-Kommando und den in Kapitel 2 eingeführten Zuständen deutlich. Dabei ist insbesondere zu beachten, daß das Kommando *do unlink* in diese Darstellung nicht sinnvoll aufgenommen werden kann, da es keinen klaren Zustandsübergang markiert.

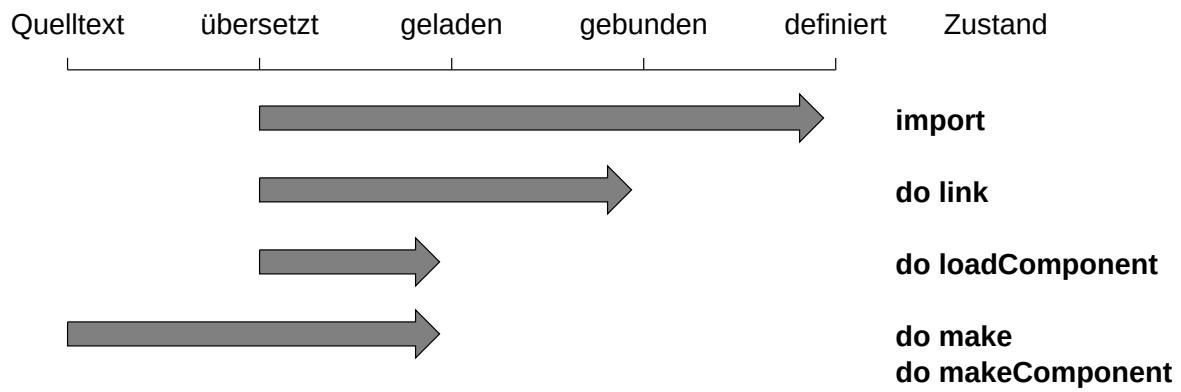

Abbildung 5.1: Zustände, in denen sich ein Modul bzw. Schnittstelle befinden kann und Kommandos, die die Zustandsübergänge auslösen

Kapitel 6

Implementierung der Tycoon Modulverwaltung

Zum gegenwärtigen Zeitpunkt ist die Modulverwaltung im Compiler des Tycoon Systems integriert. Sie wird vom Benutzer hauptsächlich über *do*-Kommandos auf dem Toplevel angesprochen (s. Ausführungen zur Semantik in Kapitel 5). Wie bereits in Abschnitt 1.2 festgestellt, beschränkt sich diese Arbeit auf die Objektidentifikation und Systemgenerierung, Tätigkeiten, die von der Modulverwaltung abgedeckt werden. Andere Aufgaben müssen, sofern überhaupt, durch externe Werkzeuge realisiert werden.

Die bisherige Analyse konzentrierte sich auf Beobachtungen von einem Standpunkt außerhalb der Modulverwaltung. In diesem Kapitel wird nun die Implementierung der Tycoon Modulverwaltung vorgestellt. Davon kann man sich im Hinblick auf die Entwicklung eines neuen Modulmanagers Hinweise und Anregungen zur Realisierung der grundlegenden Algorithmen erwarten. Zunächst wird der Aufbau der beteiligten Moduln im Detail dargestellt. Der folgende Abschnitt beschäftigt sich dann mit bestimmten wichtigen Abläufen innerhalb der Modulverwaltung. Dieses Kapitel bildet auch die Grundlage der im Rahmen der praktischen Arbeiten am Tycoon Compiler vorgenommenen Verbesserungen und Erweiterungen (s. Kapitel 7).

6.1 Architektur

Die Funktionalität der Modulverwaltung ist über drei Moduln verteilt: *tyDepend*, *tlObject*, *tlLibrary*. Ihre Beziehungen untereinander und zu anderen wichtigen Moduln des Compilers sind in Abbildung 6.1 dargestellt. Die auf den ersten Blick etwas verwirrend erscheinenden Beziehungen zwischen der Modulverwaltung und dem Rest des Compilers sind verständlich, wenn die Aufgaben der einzelnen Moduln geklärt sind. Nach den Ausführung zu den einzelnen Teilen der Modulverwaltung gehe ich kurz auf den Zusammenhang mit anderen Moduln des Compilers ein.

[1] Um die Darstellung möglichst übersichtlich zu halten, wurden Moduln und die zugehörigen Schnittstellen jeweils zusammengefaßt.

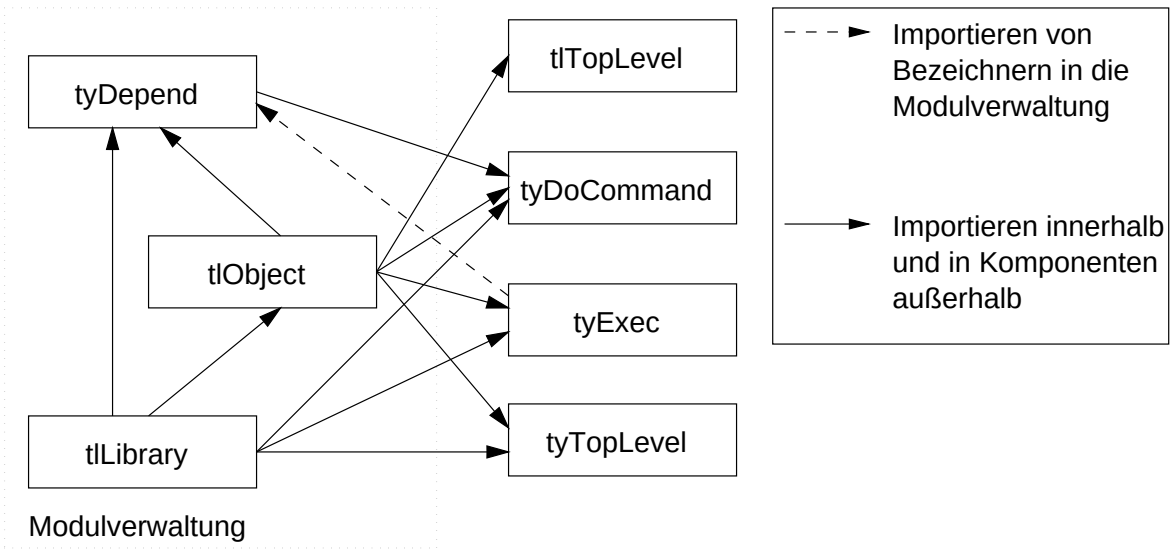

Abbildung 6.1: Importbeziehungen der Komponenten[1]der Modulverwaltung

6.1.1 tlObject

Im Modul *tlObject* werden die übersetzten Moduln und Schnittstellen verwaltet. Dies beinhaltet das Speichern und Laden auf bzw. vom Sekundärspeicher und den entsprechenden Abgleich mit dem Objektspeicher. Es werden die nach dem Importieren und Binden von Komponenten nötigen Anpassungen der Typinformation und der de-Bruijn Indizes vorgenommen. Bei diesen Vorgängen ist gewährleistet, daß die importierten Komponenten zu denen im Objektspeicher "passen", also zusammen mit diesen übersetzt wurden und nicht mit anderen Versionen dieser Objekte. Jeder Komponente ist eine Art Versionsnummer zugeordnet, die sich aus der Systemzeit ableitet. Man darf sie allerdings nicht mit einer Version im Sinne einer Versionsverwaltung wie z. B. SCCS[2] oder RCS[3] verwechseln.

Gleichfalls ist es nicht möglich, daß ein Modul mit einer geänderten Komponente arbeitet, ohne es neu zu übersetzen. Dies ist auch dann nicht möglich, wenn sich die Änderungen bzw. Erweiterungen nicht auf den von dem Modul benutzten Teil beziehen. Alte Versionen eines Moduls kann der Compiler nicht ansprechen, auch wenn diese mit Hilfe eines Versionskontrollprogramms verwaltet werden, da der Compiler stets die gleiche Datei anspricht. Diese wird von einer eventuell neu erzeugten Version überschrieben.

Zentrale Datenstruktur von *tlObject* ist zum einen *Import*, das eine kompilierte Einheit repräsentiert. Daneben kann über *Map* auf den Link-Vektor, die Wurzelbibliothek, das Verzeichnis aller im Objektspeicher befindlichen Komponenten und auf den Objektspeicher selbst zugegriffen werden. Über den Link-Vektor, ein indiziertes Verzeichnis, sind der Wert eines Moduls, sein Bytecode und seine Literale erreichbar. Nur wenn ein Modul in den Link-Vektor eingetragen ist, kann es auch referenziert werden. *Map* ist über die Schnittstelle nicht nach außen sichtbar. Zugriffsprozeduren für die Wurzelbibliothek sind deshalb nicht in *tlLibrary* sondern in *tlObject* enthalten.

[2]SCCS: "Source Code Control System".
[3]RCS: "Revision Control System".

6.1.2 tyDepend

Das Modul *tyDepend* stellt die Verbindung zwischen in Quelltextform befindlichen Moduln, Schnittstellen und Bibliotheken und deren Objektdateien und importierten Komponenten her. Dies geschieht durch die Datenstruktur *Info*, die für jede Komponente eine Liste der durch sie importierten Komponenten und Informationen über die Konsistenz von Quelltext und Objektdatei enthält. Die Abhängigkeiten zwischen den Komponenten, die durch die Importbeziehungen entstehen, werden in einem Abhängigkeitsgraphen abgebildet, der als Hash-Tabelle implementiert ist. Sie stellt eine Verbindung zwischen dem Namen einer Komponente und deren *Info*-Struktur her.

Die durch die Schnittstelle bereitgestellten Prozeduren *makeLibraries* und *make* sorgen für die Übersetzung der Bibliotheken bzw. der Schnittstellen und Moduln. Der Compiler verhindert unnötige Übersetzungen dadurch, daß der Zeitstempel von Quelltext und Objektdatei zur Überprüfung der Konsistenz herangezogen wird. Eine über dem Quelltext berechnete Prüfsumme (Fingerprint) gibt weiter darüber Aufschluß, ob der Code signifikant geändert wurde (vgl. Abschnitt 3.3). Hiermit sind Änderungen gemeint, die über Hinzufügen von Leerzeilen und Kommentaren hinausgehen.

6.1.3 tlLibrary

Tycoon bietet die Möglichkeit, Schnittstellen und Moduln in Bibliotheken zusammenzufassen (vgl. Abschnitt 2.3). Bibliotheken können wiederum in anderen Bibliotheken enthalten sein [Mat93]. Die Verwaltung der Bibliotheken geschieht im Modul *tlLibrary*. Es repräsentiert die durch die Bibliotheken aufgebaute hierarchische Struktur durch eine entsprechende Datenstruktur. Eine weitere wichtige Aufgabe besteht darin, zu prüfen, ob eine Schnittstelle oder ein Modul korrekt in der Bibliothek definiert ist. Dazu müssen alle von der Komponente importierten Schnittstellen und Moduln auch in deren Sichtbarkeitsbereich sein, also vor dieser in der Bibliothek definiert werden. Sind alle Komponenten korrekt definiert, keine Komponenten doppelt vorhanden und gibt es keine Verweise auf nicht deklarierte Komponenten, so kann die Bibliothek inklusive ihrer enthaltenen Bibliotheken übersetzt werden. Dazu wird die oben erwähnte Datenstruktur aufgebaut.

Wie auch im Modul *tlObject* für übersetzte Einheiten gibt es hier analoge Prozeduren zur Verwaltung von Bibliotheken auf Sekundärspeichern.

6.1.4 Schnittstelle zur Modulverwaltung

Abbildung 6.1 deutet an, daß die oben behandelten Moduln eine Einheit bilden. Sie werden dennoch nicht über eine einzige Schnittstelle angesprochen, was durch die Einführung eines weiteren Moduls und durch die Auflösung der Importbeziehung von *tyExec* zum Modul *tyDepend* möglich wäre. Vielmehr bleibt die Aufteilung der Aufgaben über die Moduln *tyDepend*, *tlObject*, *tlLibrary* auch in den übergeordneten Komponenten sichtbar.

Im Modul *tyDoCommand* werden die auf dem Toplevel eingegebenen *do*-Kommandos behandelt. Diese lassen sich, soweit sie die Modulverwaltung betreffen, unterteilen in die systemgenerierenden Kommandos *do make* und *do makeLibraries* (*tyDepend*), Kommandos zum Laden, Linken und Binden (*tlObject*) und zum Zugriff auf die Wurzelbibliothek (*tlLibrary*).

Das einzige Modul, das sowohl Funktionalität von der Modulverwaltung nutzt als auch bereitstellt, ist *tyExec*. Einerseits muß zum Übersetzen einer Komponente deren Sichtbarkeitsbereich geprüft werden, wozu wiederum die Wurzelbibliothek benötigt (*tlLibrary*), die

übersetzten Komponenten im Link-Vektor eingetragen und Typinformation der bereits definierten Komponenten abgefragt werden (*tlObject*). Andererseits wird das Übersetzen einer Komponente (*executeUnit* in *tyExec*) durch *do make* angestoßen (*tyDepend*). Diese Beziehung ist in Abbildung 6.1 gestrichelt dargestellt.

Eine weitere Aufgabe von *tyExec* im Zusammenhang mit der Modulverwaltung ist das Ausführen von Toplevel-Bindungen. Die hierzu notwendigen Zugriffe und Veränderungen der Typinformation werden durch das Modul *tlObject* realisiert.

Lediglich um die im initialen Toplevel enthaltenen Bezeichner zu erlangen, wird *tlObject* in *tlTopLevel* importiert. Ähnliches gilt für *tyTopLevel*, das außerdem die Wurzelbibliothek neu setzt.

6.2 Abläufe

In diesem Abschnitt werden typische Abläufe der Modulverwaltung im Detail erläutert. Ich habe mich dabei auf Abläufe beschränkt, die eine zentrale Rolle spielen:

- Das Importieren ist eine der vom Benutzer am häufigsten gewählten Aktionen im Zusammenhang mit der Modulverwaltung. Es war Gegenstand intensiver Studien im Hinblick auf eine Performanzverbesserung, die im Rahmen dieser Arbeit auch in einem gewissen Maß erreicht werden konnte. Die Suche nach neu aufgetretenen Fehlern führte immer wieder in die hier beschriebenen Bereiche, gab neue Rätsel auf, vertiefte letztlich aber das Verständnis.

- Die Systemgenerierung mit Hilfe des *do make*-Kommandos stellt ebenfalls einen der wichtigen Zugangspunkte zum Tycoon-System dar. Das Verständnis dieses Vorgangs ist gerade im Hinblick auf die praktische Arbeit besonders wichtig, da er durch eine neue Entwicklung ersetzt werden soll.

6.2.1 Importieren

Der genaue Ablauf des Importierens innerhalb der Modulverwaltung soll anhand eines konkreten Beispieles verdeutlicht werden. Ausgangspunkt ist dabei die folgende Bibliothek:

```
library Fruits
with
  interface Orange Apple Peach
  module
   apple :Apple
   peach :Peach
  interface Blueberry
  module
   blueberry :Blueberry
   orange :Orange
end
```

Der Code der Schnittstellen und Moduln ist hier nur sehr verkürzt wiedergegeben. Er beschränkt sich auf das für dieses Beispiel Wesentliche: die Importbeziehungen.

```
interface Orange
import

end

module orange
import
 apple :Apple
 blueberry :Blueberry
end
```

```
interface Apple
import

end

module apple
import

end
```

```
interface Blueberry
import
 peach :Peach
end

module blueberry
import
 peach :Peach
end
```

```
interface Peach
import

end

module peach
import

end
```

Es sei nun angenommen, daß die Bibliothek *Fruits* mit

```
do makeLibraries Fruits;
```

ggf. übersetzt und dann zur Wurzelbibliothek gemacht wurde. Anschließend wurden durch

```
import blueberry;
```

die Bezeichner *blueberry*, *Blueberry*, *peach* und *Peach* auf dem Toplevel definiert. Daraus ergeben sich die in Abbildung 6.2 dargestellten Importbeziehungen. Neben den explizit angegebenen Importbeziehungen besteht eine weitere. Es ist dies die implizit zwischen einem Modul und seiner Schnittstelle bestehende Importbeziehung, die im Quelltext nicht angegeben werden muß, aber dennoch vom Compiler erzeugt wird. In der Abbildung wurde kein zeichnerischer Unterschied zwischen Schnittstellen und Moduln gemacht. Dieser ergibt sich aber aus der gewählten und in Tycoon allgemein verwendeten Schreibweise: Schnittstellen beginnen mit einem großen Buchstaben, Moduln mit einem kleinen.

In Kapitel 5 wurden bereits die verschiedenen Zustände erläutert, in denen sich eine Komponente befinden kann. Unter Bezugnahme auf diese Zustände wird nun der Vorgang des Importierens im Detail beschrieben, wenn auf dem Toplevel

```
import orange;
```

eingegeben wird. Zur Verdeutlichung dient Abbildung 6.3.

Laden

Im ersten Schritt wird zunächst geprüft, ob sich die zu importierende Komponente überhaupt in der Wurzelbibliothek befindet. Dabei werden versteckte (*hidden*) Komponenten ignoriert

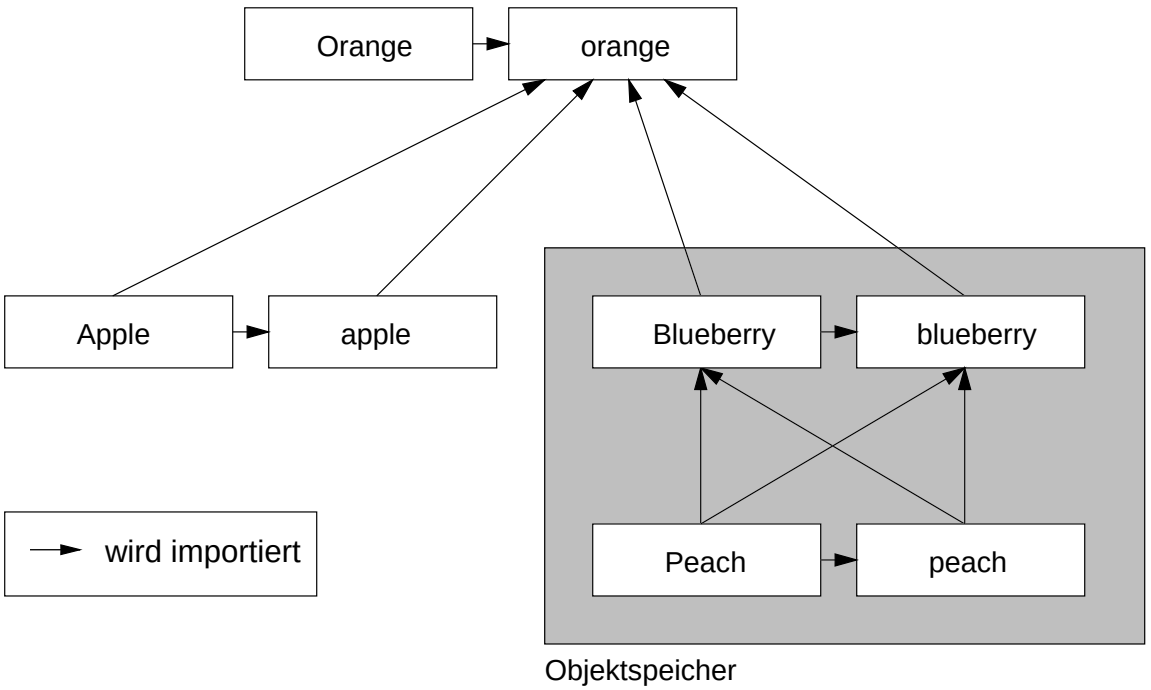

Abbildung 6.2: Importbeziehungen innerhalb der Bibliothek *Fruits*

und sichergestellt, daß es sich nicht um einen Bezeichner für eine Bibliothek handelt, der nicht importiert werden kann. Je nach Komponente, Schnittstelle oder Modul gibt die Prozedur *collectImports* die entsprechende Variante der zur Repräsentation benutzten Datenstruktur *Imports* zurück. Befand sich die Komponente nicht im Objektspeicher, so wurde sie vorher geladen. Beim Lade-Vorgang werden bereits Code und Literale eines Moduls in den Link-Vektor geschrieben. Der Modul-Wert ist in diesem Moment noch nicht bekannt.

Binden und Laden weiterer Komponenten

Es befindet sich jetzt das Modul *orange* im Objektspeicher. Sein Wert ist allerdings weder im Link-Vektor eingetragen noch sind alle Komponenten, von denen es abhängig ist, geladen. Die Importliste des Moduls *orange* enthält die Moduln *apple* und *blueberry* und die Schnittstellen *Orange*, *Apple* und *Blueberry*. Diese Importliste und alle Importlisten der dort verzeichneten Komponenten werden nun mittels Rekursion durchgegangen. Noch nicht im Objektspeicher befindliche Komponenten, in diesem Fall *Orange*, *apple* und *Apple* werden gelesen. Abhängig vom Typ der Komponente geschieht folgendes:

Modul: Ist der Zeitstempel (*ID*) der Schnittstelle des in der Importliste stehenden Moduls und des evtl. geladenen Moduls unterschiedlich, so wurde das gelesene Modul in einem anderen Kontext, mit einer anderen Schnittstelle, übersetzt. Das Modul kann also nicht gebunden werden. Es ist keine Implementation der Schnittstelle, die zum Zeitpunkt der Übersetzung des Moduls, das in der Importliste steht, im Objektspeicher war.

Fällt der Versionsvergleich positiv aus, wird, nachdem alle in der Importliste dieses Moduls stehenden Komponenten gebunden sind, das Modul selbst behandelt.

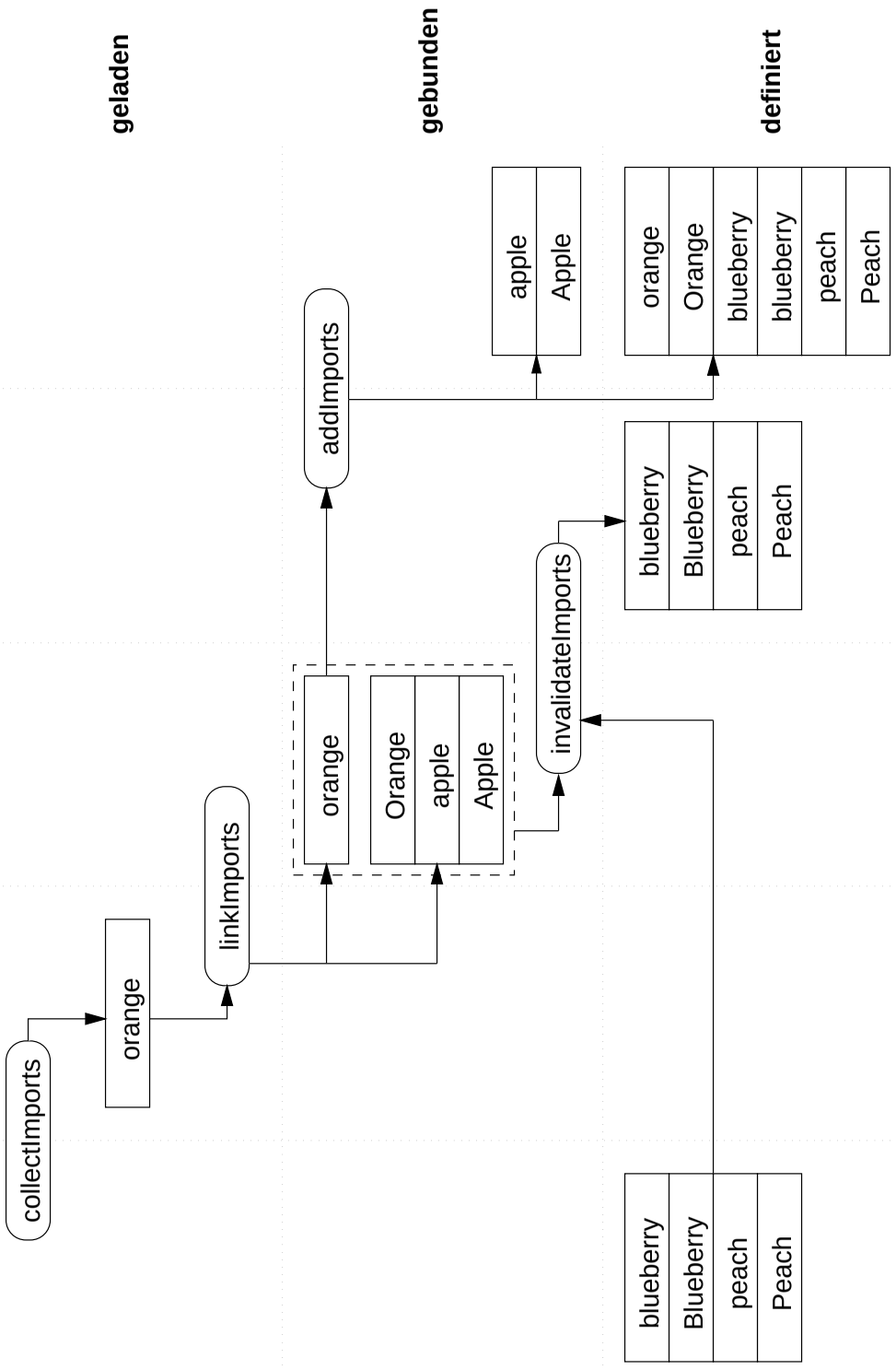

Abbildung 6.3: Import eines Bezeichners auf dem Toplevel. Darstellung der wichtigsten beteiligten Prozeduren und deren in vereinfachter Form wiedergegebenen Eingangs- und Ausgangsparameter.

Der Bindevorgang selbst besteht darin, den Modul-Code, die Literale und den Modul-Wert in den Link-Vektor einzutragen. Bevor dies getan werden kann, muß der Modul-Wert evaluiert werden. Dies erledigt die Funktion *call2* aus dem Modul *tm*. Zum Zeitpunkt, da der Code des Moduls erzeugt wurde, mußten Verweise auf referenzierte Moduln bzw. deren Werte aufgelöst werden. Jetzt, beim Binden, sind diese Verweise nicht mehr gültig, da die referenzierten Komponenten andere Indizes im Link-Vektor haben. Die Auflösung der Verweise geschieht über einen weiteren Vektor, den *Import-Vektor*, durch die Prozedur *fillImportVektor*. Dieser wird der Funktion *call2* als Argument übergeben und enthält die Zuordnung zwischen im Modul-Code verwendetem Index und aktuellem im Link-Vektor.

Schnittstelle: Nach einem Vergleich der Zeitstempel ähnlich dem im Falle eines Moduls, mit der Einschränkung, daß hier die Objekt-IDs der Schnittstellen selbst verglichen werden, wird die Bearbeitung mit der Importliste der Schnittstelle fortgesetzt. Da eine Schnittstelle nicht gebunden wird, ist die einzige Operation, die ausgeführt wird, die, die Komponente zu laden.

Zwei abstrakte Datentypen, die jeweils eine Hashtabelle verwalten, sorgen während der Bearbeitung dafür, daß bereits überprüfte Komponenten nicht mehrfach getestet und bereits gebundene nicht noch einmal dieser Prozedur unterworfen werden. Dies spart teure Sekundärspeicherzugriffe und generell Ausführungszeit.

Alle gebundenen Moduln sind in einer Liste protokolliert. In der Abbildung sind die entsprechenden Komponenten durch einen gestrichelten Rahmen gekennzeichnet. Diese Liste fungiert als eine Art Rückgabewert, auch wenn sie genaugenommen nicht das Funktionsergebnis ist. Sie wird zum Invalidieren bereits im Objektspeicher befindlicher alter Komponenten benötigt.

Alte Komponenten entfernen

Wenn beim Binden eine neue Version einer schon im Objektspeicher liegenden Komponente geladen und gebunden wurde, muß die alte aus der Liste, die alle auf dem Toplevel importierten Komponenten enthält (*env.imports*), gelöscht werden, damit die neue Version hinzufügt werden kann. Dieser Fall tritt in dem hier gewählten Beispiel nicht auf. Er kann auch nur dann auftreten, wenn zwischenzeitlich mit einem *do make*-Kommando eine Komponente übersetzt wurde. Nur dann wird beim Importieren die alte Version im Objektspeicher ungültig und kann ersetzt werden.

Müssen tatsächlich Komponenten ersetzt werden, so werden hier durch *invalidateImports* zunächst nur die alten Versionen aus der Liste aller Importe gelöscht. Genauer: Alle die Komponenten werden gelöscht, die sowohl in der durch *linkImports* protokollierten Liste (gestrichelter Rahmen) als auch auf dem Toplevel importiert sind.

Toplevel-Bindungen definieren

Im letzten Schritt müssen nun die Bezeichner auf dem Toplevel definiert werden, die explizit in der *import*-Klausel aufgeführt sind oder durch deren Schnittstellen sichtbar werden. Sie werden dazu in die schon erwähnte Liste aller Toplevel-Importe eingetragen.

Als Eingabe für die Prozedur, die dies ausführt (*addImports*), dienen die im ersten Schritt erhaltenen Komponenten (*orange* und *Orange*), da nur sie sichtbar werden sollen. Die Be-

handlung der Komponenten, die nicht sichtbar werden, war bereits nach dem Binden abgeschlossen.

In einem rekursiven Ablauf werden alle Komponenten und deren Importlisten abgearbeitet. Dabei werden wieder mehrere Fälle unterschieden und getrennt behandelt.

Modul: Bevor das Modul selbst definiert werden kann, werden seine Schnittstelle und deren importierte Komponenten gebunden. Um die daraus gewonnene neue Typumgebung und erweiterte Importliste werden die bestehende Umgebung und Importliste erweitert. Anschließend wird der Modulwert vom Link-Vektor in den Toplevel-Vektor kopiert, damit er vom Toplevel aus erreichbar ist.

Schnittstelle: Auch hier werden erst die Importe der Schnittstelle bearbeitet. Danach wird der Schnittstelle der de-Bruijn-Index 1 zugewiesen und die anderen Indizes der bereits definierten Komponenten jeweils um eins inkrementiert. Zusätzlich müssen die Indizes des Schnittstellen-Typs an die momentane Typumgebung angepaßt werden (*adjustType*). Schließlich erfolgt auch hier eine Anpassung der Typumgebung und Importliste analog zum Modul-Fall.

Eingebauter Bezeichner: In diesem Fall erfolgt keine Bearbeitung. Eingebaute Bezeichner sind zwar in der Importliste enthalten, aber schon auf dem Toplevel definiert.

Nicht berücksichtigt in dieser Beschreibung wurde die Behandlung von Fehlern. Dies hätte den Umfang und die Verständlichkeit negativ beeinflußt.

6.2.2 Übersetzung

Als Ausgangspunkt dient das im vorigen Abschnitt eingeführte Beispiel mit den in Abbildung 6.2 gezeigten Abhängigkeiten. Es soll angenommen werden, daß der Quelltext der Schnittstelle *Blueberry* und seine Implementation *blueberry* so geändert wurde, daß er einer Übersetzung bedarf.

Der grundsätzliche Algorithmus ist einfach und schreibt vor, bevor eine Komponente übersetzt werden kann, zunächst alle transitiv erreichbaren Komponenten zu übersetzen. Dieser Ablauf wird von den wechselseitig rekursiven Prozeduren *makeUnit* und *makeImports* aus dem Modul *tyDepend* implementiert.

In *makeUnit* wird die entsprechende Komponente gelesen und getestet, ob eine Übersetzung nötig ist. Dies erfolgt nach den Kriterien aus Abschnitt 5.2. Bevor die Komponente selbst übersetzt wird, werden in *makeImports* alle von ihr importierten Komponenten behandelt. Zu diesen importierten Komponenten wird bei einem Modul auch die entsprechende Schnittstelle hinzugefügt. In unserem konkreten Beispiel bedeutet dies: Nach Eingabe von

```
do make blueberry;
```

wird zunächst das übersetzte Modul *blueberry* gelesen und festgestellt, daß es mit seinem Quelltext nicht mehr konsistent ist. Zu der Liste seiner Importe wird dann die Schnittstelle *Blueberry* hinzugefügt. Diese Liste wird nun in gleicher Weise abgearbeitet.

Sind alle Importe einer Komponente übersetzt, so wird der eigentliche Übersetzungsvorgang eingeleitet. Er ist nicht mehr Teil der Modulverwaltung und erfolgt daher im Modul *tyExec*.

Um zu verhindern, daß Komponenten mehrfach bearbeitet werden, sind die bereits behandelten in einer Hashtabelle gespeichert. Obwohl sie den Namen *graph* führt, handelt es sich nicht um einen Graphen oder sogar Abhängigkeitsgraphen in herkömmlichem Sinn. Die Abhängigkeiten werden durch die Importlisten beschrieben. Die Hashtabelle stellt keine Verbindung zwischen Komponenten her.

Kapitel 7

Erweiterungen und Verbesserungen

Die Schilderung der Realisierung der Modulverwaltung in Kapitel 6 bildet nicht nur die Voraussetzung für die Entwicklung eines neuen Modulmanagers. Sie ist auch Grundlage für Veränderungen an der existierenden Implementierung. Zu diesen Veränderungen zählt zum einen die Beseitigung von Fehlern und zum anderen Verbesserungen im Laufzeitverahlten.

Im Rahmen der praktischen Arbeiten sind die im folgenden dokumentierten Änderungen am Tycoon-Compiler vorgenommen worden. Sie haben Eingang in die allgemeine Entwicklung gefunden.

7.1 Link-Vektor

Der Link-Vektor wurde bisher im Modul *tyRuntime* als Array verwaltet. Jeweils drei aufeinanderfolgende Einträge gehörten zu einem Modul. Es sind die schon mehrfach erwähnten Einträge für Code, Literale und evaluierten Wert eines Moduls.

Da der Link-Vektor zur Modulverwaltung gehört, erscheint eine Verlegung der kompletten Link-Vektor-Verwaltung vom Modul *tyRuntime* in das Modul *tlObject* logisch. Somit werden alle Aktivitäten des Link-Mechanismus in *tlObject* behandelt. Auch auf eine explizit persistente Speicherung des Link-Vektors, wie bisher geschehen, kann verzichtet werden, da dies bei einem Aufruf von *do saveSystem* auf dem Toplevel ohnehin geschieht.

Folgendermaßen wurde vorgegangen: Im Modul *tlObject* wurde ein neuer Typ *LinkTableIndex* angelegt, der die Einträge eines Link-Vektorelements aufnimmt. Bei den Einträgen handelt es sich um *tm.OID*s, die die entsprechenden Verweise auf den Code, die Literale und den Wert eines Moduls enthalten. Als Datentyp für den eigentlichen Vektor wurde ein dynamisches Array gewählt, einerseits um die Elemente des Vektors über Indizes ansprechen zu können. Andererseits ist es möglich, die Zunahme der Größe des Link-Vektors zur Laufzeit zu berücksichtigen, ohne wie bei einem festen Array darauf angewiesen zu sein, einen ausreichend großen Anfangswert zu schätzen und dabei Gefahr zu laufen, daß dieser irgendwann nicht mehr ausreicht. Auch scheint es bei der Modellierungsmächtigkeit der Sprache TL nicht angemessen zu sein, die Position des gewünschten Wertes über Addition von Index und konstantem Offset für die entsprechende Größe zu berechnen.

Bedingt durch den neuen Datentyp und die damit verbundene Änderung der Verwaltung der Einträge im Link-Vektor mußte dieser Teil entsprechend geändert werden. Es wurden drei neue Funktionen eingeführt, die die jeweiligen Werte aus dem Link-Vektor extrahieren: *getBytecode*, *getLiterals* und *getValue*

```
library Root
with
  library
   stdenv
   bulkenv
   TestLib
end

library TestLib
import
with
  interface
   Test
  module
   test :Test
end
```

Abbildung 7.1: Beispiel ineinandergeschachtelter Bibliotheken

Die Verwaltung der freien Einträge im Link-Vektor muß nun nicht mehr explizit vorgenommen werden, sondern kann vom Modul *dynArray* übernommen werden, das diese Funktionalität schon bereitstellt. Auch die Funktion *setI*, in der zunächst ein eventuell schon vorhandener Eintrag im Link-Vektor gelöscht wird, bevor derselbe mit *allocateLinkTableIndex* wieder angefordert wird, kann nahezu unverändert bleiben. Die Zusicherung, daß ein nach einem Aufruf von *freeLinkTableIndex* angeforderter Index exakt der zuvor freigegebene Index ist, kann aufgrund der Angaben in der Schnittstelle zu *dynArray* gegeben werden. Damit ist sichergestellt, daß eine neue Version einer schon im Objektspeicher befindlichen Komponente an die Stelle der alten kommt.

7.2 Übersetzen von Bibliotheken

Beim Absetzen eines *do makeLibraries lib*-Kommandos überprüfte der Compiler bisher, ob eine der Subbibliotheken von *lib* übersetzt werden muß. Wie bereits im Abschnitt 5.4 ausgeführt, beschränkt sich die Überprüfung auf folgendes:

- Gibt es bereits eine übersetzte Bibliothek?
- Gibt es den Quelltext zur Bibliothek?
- Sind der Fingerprint des Quelltextes und des ausführbaren Codes gleich?

Als Ausgangspunkt soll das Beispiel aus Abbildung 7.1 dienen. Die Bibliothek *Root* selbst wird nur übersetzt, wenn eine der geschachtelten Bibliotheken geändert wurde oder eine der obigen Bedingungen falsch ist. Wurde nun die Bibliothek *TestLib*, die sich in der Bibliothek *Root* befindet, zwischenzeitlich z.B. durch ein

```
do ''TestLib/TestLib.tl'';
```

ebenfalls übersetzt ohne sie jedoch zur Wurzelbibliothek zu machen, so konnte dies bei einem erneuten

```
do makeLibraries Root;
```

nicht erkannt werden. Wurden z. B. in *TestLib* neue Schnittstellen und Moduln eingeführt, so können diese jetzt nicht importiert werden.

Zur Lösung dieses Problems sollte eine zusätzliche Überprüfung in der Funktion *makeLibrary* im Modul *tyDepend* erfolgen, ob *TestLib* zeitlich nach *Root* übersetzt wurde, wenn alle oben genannten Bedingungen wahr sind. Dies kann durch entsprechenden Abgleich der Object-IDs beider Bibliotheken erfolgen, die einen Zeitstempel enthalten. Die Objekt-IDs sind in diesem Zusammenhang nicht mit den OIDs zu verwechseln, die Referenzen innerhalb des Objektspeichers darstellen. Ein Übersetzen von *TestLib* ist nicht erforderlich; es muß lediglich eine Propagierung dieser Tatsache erfolgen. Dafür bietet sich *makeState.changes* an, welches nur inkrementiert werden muß, um eine Neuübersetzung der einschließenden Bibliothek zu veranlassen. Da die Objekt-ID der einschließenden Bibliothek in *makeLibrary* nicht direkt verfügbar ist, muß sie explizit der Funktion übergeben werden. Dies erfordert eine entsprechende Anpassung aller Aufrufe von *makeLibrary* und von *compileLibrary*, da hier *makeLibrary* rekursiv aufgerufen wird.

7.3 Übersetzen von versteckten Komponenten

In Abschnitt 2.4 ist beschrieben, wie mit der *hide*-Klausel einzelne Bezeichner einer Bibliothek nach außen unerreichbar gemacht werden können. Die Akzeptanz dieser Möglichkeit ist im Bereich der Anwender und Programmierer des Tycoon-Systems äußerst gering[1]. Nur so ist es zu erklären, daß erst jetzt bemerkt wurde, daß mit versteckten Komponenten praktisch nicht gearbeitet werden kann. Dies kommt daher, daß der Compiler den Versuch, ein verstecktes Modul oder eine versteckte Schnittstelle mit *do makeComponent* bzw. *do make* zu übersetzen, mit einer Fehlermeldung abbricht. Nur mit Hilfe eines Kniffs[2] kann dies vermieden werden. An dieser Stelle soll nicht über Sinn und Zweck des *hide*-Konstruktes diskutiert werden, vielmehr sollen die Ursache des Fehlers analysiert und die zu seiner Beseitigung vorgenommenen Maßnahmen beschrieben werden.

In einer durch das Modul *tlLibrary* gepflegten Datenstruktur wird der Aufbau der Bibliotheksstruktur repräsentiert. Dabei werden die Fälle *moduleCase*, *interfaceCase*, *libraryCase* und *hiddenCase* unterschieden, also alle Komponenten, die in einer Bibliothek vorkommen können.

Der oben beschriebene Fehler tritt auf, wenn für eine versteckte Komponente zum Zweck des Übersetzens die Namen der Quelltext-Datei und der Objekt-Datei bestimmt werden sollen. Dies erledigt die Funktion *inspect* im Modul *tyDepend*. Dort wird der Fall einer versteckten Komponente jedoch nicht berücksichtigt. Neben dieser weiteren Fallunterscheidung muß für den Fall *hiddenCase* jedoch weiterhin geprüft werden, was für eine Komponente versteckt ist. Die *hide*-Klausel kann sowohl bei Moduln als auch bei Schnittstellen Anwendung finden.

[1] In keiner Bibliothek, die zum Tycoon-Compiler gehört oder die sonst im Verzeichnis *bootlib* zu finden ist, wird die *hide*-Klausel benutzt.

[2] Eine Komponente, die versteckt werden soll, muß zunächst als nicht versteckt übersetzt werden und danach in der Bibliothek unter *hide* aufgeführt werden. Dies ist aber zweifelsohne keine vertretbare Methode, versteckte Bezeichner zu benutzen.

Die eigentliche Komponente ist im Falle der *hiddenCase*-Variante im Tupelfeld *component* abgelegt. Diese muß dann zur Ermittlung der Dateinamen herangezogen werden.

Um unnötige Codeduplizierung zu vermeiden, wurde die eigentliche Bestimmung der Dateinamen, jeweils getrennt nach Modul und Schnittstelle, in die Funktionen *moduleFileNames* und *interfaceFileNames* ausgelagert.

7.4 Verzeichnis aller geladenen Komponenten

Im Abschnitt 6.1.1 wurde die Funktion der Datenstruktur *Map* bereits erläutert. Der Eintrag *dictionary* verzeichnet alle im Objektspeicher befindlichen Komponenten. Er wird benutzt, um das Vorhandensein einer Komponente zu testen oder um diese zurückzugewinnen.

Eine Beobachtung der Zugriffe auf *map.dictionary* ergab, daß bei zunächst leerem Objektspeicher z. B. ein

```
import print;
```

zu ca. 550 Zugriffen führt. Dabei werden 26 Komponenten geladen und sieben definiert. Ein

```
import tycoon;
```

zieht sogar über 5000 Zugriffe nach sich. Da bei großen Applikationen oder hochabstrakten Bibliotheken die Anzahl der Moduln, die importiert werden, stetig steigt, wird die pro Zugriff benötigte Zeit immer größer.

Weil auf *map.dictionary* nur Zugriffe stattfinden, die ein einziges Element zurückgewinnen oder auf das Vorhandensein eines Elementes überprüfen, scheint die Implementation als Dictionary nicht adäquat. Die Implementierung von dem Modul *Dictionary* stützt sich auf die Datenstruktur einer Liste, d. h. zum Durchsuchen nach einem Element muß durchschnittlich die Hälfte aller Elemente verglichen werden. Im Laufe der Zeit steigt die Anzahl der importierten Moduln stetig, und die Zeit, die pro Suche benötigt wird, nimmt somit linear zu. Ein Importieren eines Moduls ist deshalb bei schon gefülltem Objektspeicher zugriffsintensiver als bei leerem.

Als Lösung bietet sich an, das Dictionary durch eine Hashtabelle zu ersetzen. Dies hat den Vorteil, daß im Idealfall, wenn das gesuchte Element gerade an der berechneten Stelle ist, die pro Suche benötigte Zeit konstant ist. Leider ist dies nicht immer der Fall, da der berechnete Eintrag bereits gefüllt sein kann. Bei einer Kollision kann dann der nächste freie Eintrag z. B. durch sequentielle Suche ermittelt werden (*linear probing*). Auch wenn dieser Ansatz Schwächen gegenüber *double hashing* hat, so ist er doch einfach zu implementieren. Beim *double hashing* wird bei einer Kollision nicht ab der gefundenen Position ein freier Eintrag gesucht, sondern durch eine weitere Hashfunktion ein Offset berechnet. Dadurch kann das Problem des *clustering* effektiver verhindert werden.

Nachteilig an einer Hashtabelle ist ganz allgemein der im Vergleich zur Liste höhere Platzbedarf. Dies liegt zum einen daran, daß mit einem Feld fester Größe gearbeitet wird, zum anderen nimmt die Trefferquote bei einer mehr als 90% gefüllten Hashtabelle stark ab bzw. schließt sich eine weitere sequentielle Suche über mehrere Elemente an. Um optimale Performanz zu erzielen, sollten also immer genügend freie Elemente vorgehalten werden [Sed88].

Bei der vorliegenden Implementierung der Hashtabelle wurde *linear probing* verwendet. Die Tabelle wird dabei solange gefüllt, bis sie vollständig gefüllt ist. Dies stellt zwar aus den

genannten Gründen nicht die optimale Lösung dar, bringt aber gegenüber der Implementierung als Liste deutlich spürbare Vorteile.

Das Ersetzen der einzelnen Aufrufe geschieht bis auf eine Ausnahme schematisch. Beim Anlegen der Hashtabelle müssen zusätzlich eine Vergleichsfunktion für die Schlüssel, in diesem Fall vom Typ *String*, und eine entsprechende Hashfunktion, die den Schlüsselwert auf einen Index abbildet, übergeben werden. Hier wurde auf Funktionen aus dem Modul *string* zurückgegriffen, die speziell für diesen Zweck bereitgestellt werden.

Die praktische Durchführung hat dann gezeigt, daß ein `dictionary.set` in ein

```
try
  hashTable.insert (...)
when hashTable.error then
  hashTable.delete (...)
  hashTable.insert (...)
end
```

umgewandelt werden muß, um unerwünschte Fehler beim Ersetzen eines bereits vorhandenen Elementes zu verhindern. Dies muß anders als bei Benutzung des Moduls *dictionary* explizit abgefangen werden.

7.5 Synchronisieren des Objektspeichers

Bisher ist es nicht möglich, eine ausführbare Datei (*.x) , welche aktueller ist als der im Objektspeicher befindliche Code bzw. bei einer Schnittstelle die entsprechend übersetzte Typinformation, in den Objektspeicher ohne erneutes *do make* oder *do makeComponent* zu laden. Dies kann aber nötig sein, wenn nach einem *do make* bzw. *do makeComponent* und vor *do saveSystem* ein Systemabsturz auftritt. Die Hauptursache dafür, daß dies bisher nicht funktioniert, liegt darin, daß bei einem Aufruf von *linkImports* (sowohl von *topLevelImport* als auch von *link*) aus Performanzgründen im Modul *tlObject* nur geprüft wird, ob sich die entsprechende Komponente im Objektspeicher befindet, nicht jedoch, ob die Version auf dem Sekundärspeicher aktueller ist. Damit wurde auch ein durchaus vernünftiger Einsatz verfolgt. Zugriffe auf externen Speicher sind im Vergleich zu Objektspeicherzugriffen, die meist innerhalb des Hauptspeichers erfolgen können, erheblich teurer.

Benötigt wird also ein Kommando, welches explizit die entsprechende Komponente auf dem Sekundärspeicher prüft und bei Bedarf liest. Diese Aufgabe soll das neu eingebaute *do syncComponent* erfüllen.

Es hat ähnliche syntaktische Eigenschaften wie *do makeComponent*. Es kann mit mehreren Argumenten aufgerufen werden, die der Reihe nach abgearbeitet werden. Bei einem Modul oder einer Schnittstelle werden alle transitiv erreichbaren Komponenten synchronisiert. Handelt es sich bei dem Argument um einen Bibliotheksbezeichner, so werden in der Bibliothek enthaltene Komponenten aktualisiert.

Zur Realisierung werden zwei neue Funktionen eingefügt. Die erste Funktion (*synchronize*) wird vom Modul *tyDoCommand* aufgerufen und steuert den gesamten Ablauf des Synchronisierens. Es wird dann durch die zweite neue Funktion (*loadAll*) explizit jede von der betreffenden Komponente abhängige Komponente auf Konsistenz mit der im Objektspeicher befindlichen Version überprüft und bei Bedarf geladen. Ein nachfolgender Aufruf von *linkImports* führt dazu, daß alle entsprechend neuen Komponenten neu gebunden werden.

Durch eine weitere Änderung in *linkImport*, die verhindert, daß in *addImports* die falschen Schnittstellen gefunden werden, werden sogar alle abhängigen Schnittstellen gelinkt, bzw. in *info.changes* protokolliert und im anschließenden *addImports* neu definiert. Dadurch entsteht der Nachteil, daß auch bei einem normalen Import Meldungen ausgegeben werden, die schon gebundene Komponenten als neu gebunden ausweisen. Das hat zwar keinen Einfluß auf die Funktionsfähigkeit, spiegelt den tatsächlichen Vorgang aber nicht korrekt wider. Abhilfe würde eine Änderung der Datenstruktur *tlObject.T* schaffen, wo durch ein weiteres Flag angezeigt werden könnte, ob eine Komponente neu gebunden werden muß. Da aber das Sekundärspeicher-Layout einer übersetzten Datei von dieser Datenstruktur abhängt, müßte eine Konvertierung all dieser Dateien vorgenommen werden.

Eine weitere Möglichkeit, die überflüssigen Meldungen zu vermeiden besteht darin, das Sekundärspeicherlayout unabhängig von der Datenstruktur *tlObject.T* zu gestalten. Da das einzuführende Flag erst nach dem Laden in den Objektspeicher eine Bedeutung erlangt, müßte es nicht explizit gespeichert werden. Auch andere Elemente der Datenstruktur *tlObject.T* sollten in diesem Zusammenhang darauf überprüft werden, ob sie auf Sekundärspeicher abgelegt werden müssen. Die praktische Durchführung müßte in den Funktionen *loadFile* und *saveFile* im Modul *tlObject* eine Projektion der Datenstruktur *tlObject.T* auf die Struktur des Sekundärspeicherlayouts bzw. umgekehrt durchführen.

Kapitel 8

Generische und typsichere Systemgenerierung

Die Modulverwaltung in ihrer bisherigen Form umfaßt einige Dienste und Aufgaben, die in generalisierter Form auch in anderen Bereichen Anwendung finden könnten. Z. B. ist es denkbar, Objekte einer Datenbank, die in Beziehungen zueinander stehen, so zu modellieren, daß Aktualisierungen abhängig von bestimmten Kriterien durchgeführt werden. Die Beobachtung, daß zwischen Objekten Abhängigkeiten bestehen und daß gewisse Eigenschaften dieser oder anderer Objekte zu bestimmten Aktionen führen, waren schon 1979 Anlaß für die Realisierung eines Werkzeuges, das diese Aufgaben automatisch erledigt [Fel79]. Auch für diese Arbeit war es der Ausgangspunkt für die praktische Implementierungsarbeit, die hier einen ersten Schritt auf dem Weg zu einem wiederverwendbaren Modulmanager markiert.

Der in Abschnitt 6.2.2 beschriebene Algorithmus zur Bestimmung von Beziehungen zwischen Schnittstellen und Moduln und zum Auslösen der Übersetzung dieser Komponenten in Abhängigkeit der Modifikationszeit und des Fingerprints beinhaltet ebenfalls das geschilderte Generalisierungspotential. Gegenwärtig sind die Objekte, Funktionen und die Art und Weise der Bestimmung von Abhängigkeiten allerdings fest im Code der Modulverwaltung eingebaut. Die Ausführungen im Abschnitt 1.2.2 haben gezeigt, daß die Anforderungen an eine Systemgenerierung aber durchaus vielschichtiger sind. Bezogen auf die dort entwickelten Anforderungen zeigt die Modulverwaltung besonders in den folgenden Punkten Schwächen. Diese rühren allerdings daher, daß der Entwurf der Modulverwaltung die Generizität und Vielschichtigkeit dieser Anforderung aufgrund des klar umrissenen Einsatzgebietes innerhalb des Tycoon-Compilers nicht zu berücksichtigen brauchte (s. Kapitel 4).

- Die Modulverwaltung ist bzgl. der zu behandelnden Objekte auf Schnittstellen, Moduln und Bibliotheken des Tycoon-Systems und damit auf Dateien beschränkt.
- Die Bestimmung der Abhängigkeiten beruht ausschließlich auf der *import*-Klausel. Feiner abgestufte Beziehungen werden nicht unterstützt.

Ausgehend von der Schilderung in Abschnitt 4.1 erschien es nun sinnvoll, die Funktionalität der Systemgenerierung in ein nicht direkt in den Compiler integriertes, wohl aber von ihm benutztes Modul auszulagern. Der Begriff der Systemgenerierung kann damit viel weiter als in Abschnitt 1.2.2 gefaßt werden. Er schließt beliebige Systeme bestehend aus in Beziehung stehenden Objekten ein, wie z. B. Datenbanken.

Im folgenden soll zuerst aufgezeigt werden, welche Ansätze zur Bewältigung dieser Aufgabe existieren. Diese Erläuterungen werden anhand des in der Praxis häufig eingesetzten UNIX-Make vertieft. Anschließend wird der im Rahmen dieser Arbeit gewählte Ansatz vorgestellt und einer kritischen Betrachtung unterzogen.

8.1 Ansätze

Die Systemgenerierung ist nach Abschnitt 4.1 die Steuerung eines Transformationsvorganges. Zentrales Problem ist dabei die Modellierung der Beziehungen von unterschiedlichen Objekten, um die Transformationen korrekt und effizient durchführen zu können. Dazu ist es nötig, die Struktur dieser Objekte zu analysieren, d. h. festzustellen, wo die Abhängigkeitsinformation lokalisiert werden kann. Sie kann direkt in den Objekten enthalten sein oder explizit beschrieben werden. Weiterhin zu beachten ist die Abhängigkeitsrelation, die als Abhängigkeitsgraph visualisiert werden kann. Kann bzw. darf er Zyklen enthalten?

Zunächst ein Überblick der von anderen Systemen gewählten Ansätze zur Beschreibung von Abhängigkeiten [Est88]:

- Die Abhängigkeiten sind explizit in der Syntax der Sprache enthalten, wie z. B. bei Ada und Modula. Die speziell auf die Sprache zugeschnittenen Umgebungen und Compiler nutzen diese Information direkt.

- Aus dem Quelltext werden die Abhängigkeiten automatisch mittels eines Werkzeuges generiert. Der Unterschied zum ersten Punkt ist, daß hier die Abhängigkeitsinformation in expliziter Form vorliegt bzw. generiert wird. Sie kann somit von sprachunabhängigen Werkzeugen genutzt werden.

- Beziehungen werden extern bereitgestellt. Dies kann durch Sprachen (*module interconnecting languages*) wie z. B. Odin [Cle88] geschehen. Eine weitere Möglichkeit ist die Beschreibung mittels Regeln, wie sie vom UNIX-Make verwendet wird. Kern sowohl von Odin als auch von Make ist der Abhängigkeitsgraph. Die Spezifikationssprache Odin zeichnet sich darüber hinaus dadurch aus, daß sie mehrere Typen von Abhängigkeiten unterstützt.

Die vorgestellten Lösungen überschneiden sich zu einem gewissen Teil und werden in der Praxis teilweise gleichzeitig genutzt. Wichtig ist, daß sie sich mit einer Ausnahme alle ausschließlich auf Programmiersprachen beziehen und damit in bezug auf die von mir postulierte Generizität der zu behandelnden Objekte bereits eine Einschränkung darstellen. Eine Modellierung von Objekten und deren Beziehungen innerhalb einer Datenbank ist mit diesen Ansätzen z. B. nicht zu erreichen. Nur in einem Makefile lassen sich beliebige Objekte aufführen, dafür ist die Granularität der Objekte auf Dateigröße beschränkt.

Aus dieser Aufstellung wird weiterhin deutlich:

1. Die Art der Objekte ist fest vorgegeben. Meist sind sie auf Dateien bzw. bestimmte Dateitypen eingeschränkt.

2. Über das Auftreten der Abhängigkeitsinformation sind feste Annahmen getroffen worden. In den aufgeführten Fällen muß diese Information entweder explizit vom Anwender bereitgestellt oder vom System generiert werden.

8.2 UNIX-Make

Im folgenden wird ein in der Praxis verwendeter Vertreter der Systemgenerierung vorgestellt, das *Unix-Make*[1] Mit dem UNIX-Make können Beziehungen zwischen Dateien modelliert werden. Zur Beschreibung dieser Beziehungen bzw. Abhängigkeiten und der auszuführenden Anweisungen verwendet es Regeln (*rules*), die in einer speziellen Datei, dem *Makefile*, abgelegt sind, also extern bereitgestellt werden. Eine über die Grenzen der Datei hinausgehende feinere Beschreibung der Abhängigkeiten, z.B. auf Prozedur-Ebene ist nicht möglich.

8.2.1 Regeln

Die allgemeine Form einer Regel ist folgende:

```
targets: dependencies
   commands
```

Dabei werden die Anweisungen *commands* genau dann ausgeführt, wenn mindestens eine der Dateien in *targets* in bezug auf *dependencies* nicht mehr aktuell ist oder eine der Dateien in *targets* noch nicht existiert. *Make* stellt dies anhand des Zeitstempels der Datei fest. Eine Datei kann immer nur in einem *targets* auftreten, das auch eine *commands*-Zeile hat, so daß alle *dependencies* zusammengefaßt werden können.

Eine Ausnahme bilden die sogenannten *double-colon*-Regeln, die in der Praxis nur sehr selten benutzt werden. Diese erlauben es, eine Datei von mehreren voneinander disjunkten Mengen von Dateien abhängig zu machen. Die allgemeine Form lautet analog zu oben:

```
targets:: dependencies
   commands
```

Eine Datei darf entweder nur in *targets* dieser Art oder der obigen Art verzeichnet sein, ein Mischen ist nicht erlaubt.

8.2.2 Implizite Regeln

Für eine Reihe von Abhängigkeiten existieren bereits vordefinierte Regeln. So ist es z. B. nicht nötig, explizit die Regel zur Erzeugung einer Objekt-Datei aus einem C-Quelltext hinzuschreiben. Hierfür gibt es eine implizite Regel, die aufgrund der Dateiendung (.o) und einer fehlenden Regel zu der entsprechenden Datei bestimmt, daß es sich hierbei um die Erzeugung einer Objekt-Datei handelt. Dazu wird in diesem Fall versucht, eine gleichnamige Datei mit der Endung .c zu finden und diese mit Hilfe des Compilers (cc) zu übersetzen.

Implizite Regeln dienen ausschließlich zur Vereinfachung von Makefile-Dateien. Sie führen keine neuen Konzepte ein oder bringen mehr Mächtigkeit. Insbesondere definieren sie den Typ einer Datei ausschließlich über deren Endung und nicht über den Inhalt, der völlig unabhängig vom Namen gewählt werden kann.

[1] Im folgenden wird das meist im Umfeld des UNIX Betriebssystems eingesetzte Programm *Make* kurz mit *Unix-Make* bezeichnet.

8.2.3 Abhängigkeiten und Datenfluß

Aus einer Reihe von Regeln lassen sich zwei unterschiedliche Darstellungen gewinnen. Dies sei an dem folgenden, aus [Fel79] entnommenen, Beispiel erläutert, das das Makefile eines einfachen Compilers zeigt:

```
pgm: codegen.o parser.o library
  cc codegen.o parser.o library -o pgm   #load

codegen.o: codegen.c definitions
  cc -c codegen.c        # compile

parser.o: parser.c definitions
  cc -c parser.c         # compile

parser.c: parser.y
  yacc parser.y          # generate parser into file y.tab.c
  mv y.tab.c parser.c    # change name of output
```

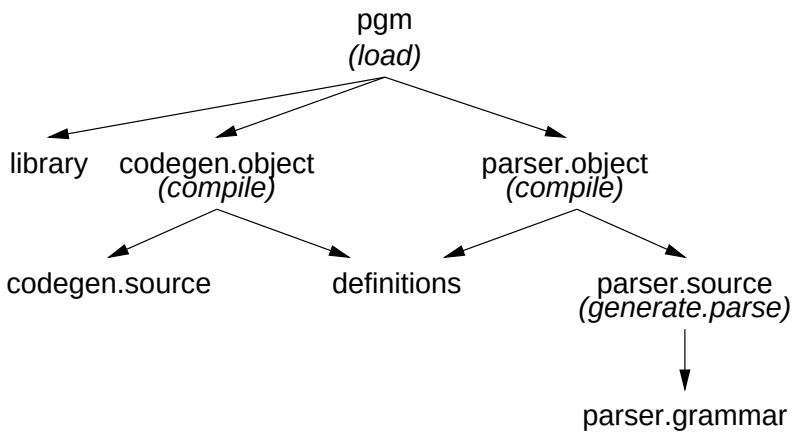

Abbildung 8.1: Abhängigkeitsgraph. Darstellung entnommen aus [Fel79].

Zum einen ergibt sich aus den *targets: dependencies*-Zeilen der Abhängigkeitsgraph (Abbildung 8.1). In diesem können optional die Kommandos zur Erzeugung der Knoten (*targets*) hinzugefügt werden.

Aus den *commands*-Zeilen des Makefile ergibt sich der in Abbildung 8.2 gezeigte Datenflußgraph. Dieser zeigt an, welche Dateien von welchen Programmen in welche anderen Dateien transformiert werden. Z. B. wird durch

```
cc -c codegen.c
```

der C-Quelltext *codegen.c* durch den C-Compiler in die Objectdatei *codegen.o* überführt.

Im idealen Fall kann der Abhängigkeitsgraph aus dem Datenflußgraph gewonnen werden. Dazu wird die Richtung der Pfeile invertiert, und die Kommandos werden an die entsprechenden Knoten (*targets*) geschrieben. Dieser Vorgang ist umkehrbar. Das Verfahren ist jedoch nur dann anwendbar bzw. führt zum richtigen Ergebnis, wenn

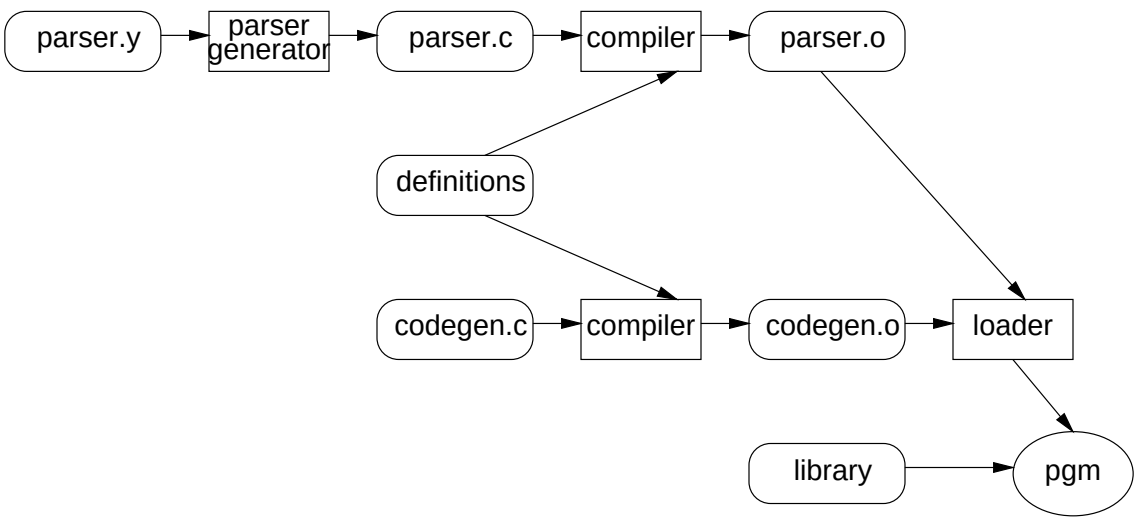

Abbildung 8.2: Datenflußraph. Darstellung entnommen aus [Fel79].

- das Kommando alle in der Abhängigkeitsliste aufgeführten Dateien auch zum Ausführen benötigt und
- das Kommando genau und nur die *targets*-Datei erzeugt.

Dem Benutzer ist also freigestellt, ob er seine Abhängigkeiten so formuliert, daß Datenfluß- und Abhängigkeitsgraph auseinander hervorgehen können. Er hat andererseits auch keine Möglichkeit, dies zu fordern bzw. zu erzwingen, um z. B. Fehler zu vermeiden o.ä. Dies ist insbesondere dann ein gravierender Nachteil, wenn die Makefiles nicht mit einem Werkzeug automatisch erstellt werden. Objekte einerseits und Makefiles andererseits müßten parallel gewartet werden, was zu Inkonsistenzen führen kann.

8.2.4 Automatische Abhängigkeiten

Die Bestimmung der Abhängigkeiten eines größeren Softwareprojektes ist, wenn sie "von Hand" vorgenommen werden muß, eine fehlerintensive Arbeit. *Make* bietet keine Möglichkeiten, um diesen Vorgang zu automatisieren. Es existieren daher z. B. für C-Dateien einige Lösungen, die auf der Basis von *sed*-Skripten oder mit Hilfe des *makedepend*-Programms aus dem *Imake*-Paket diese Abhängigkeiten automatisch erzeugen [DuB93]. Eine weitere Möglichkeit bietet *mmf* [BV88]. Hier können die Abhängigkeiten auch auf Basis der tatsächlich benutzten Bezeichner ermittelt werden. Dies geschieht durch Untersuchung der Objektdateien auf Referenzen, die auf externe Objekte verweisen.

8.3 Systemgenerierung in TL

Es können nun ausgehend von der Schilderung möglicher Ansätze in Abschnitt 8.1 und der Beschreibung des UNIX-Make in Abschnitt 8.2 Anforderungen an eine generische Systemgenerierung formuliert werden:

- Neben der Behandlung von Dateien soll es ebenfalls möglich sein, beliebige Objekte zu beschreiben.

- Jedem behandelten Objekt sollen spezifische Methoden z. B. zur Transformierung zugeordnet werden können.
- Die Formulierung von Abhängigkeiten soll nicht auf bestimmte Modelle eingeschränkt sein. Der durch die Abhängigkeitsrelation gebildete Abhängigkeitsgraph soll z. B. auch Zyklen enthalten dürfen.
- Zwischen Objekten können verschiedene Abhängigkeiten bestehen. Zwischen Moduln und Schnittstellen besteht z. B. eine *Import*-Beziehung, während zwischen Bibliotheken eine *Enthalten*-Beziehung bestehen kann. Ein generischer Ansatz soll beliebige Arten von Abhängigkeiten unterstützen.

8.3.1 Repräsentation

Um die im vorigen Abschnitt entwickelten Anforderungen zu erfüllen, muß zunächst die Frage beantwortet werden, wie die Abhängigkeitsinformation repräsentiert werden kann. Dazu ist es in einem ersten Schritt nötig, diese Information bereitzustellen. Da die Struktur der zu betrachtenden Objekte bei einem generischen Ansatz unbekannt ist, kann diese Information nur vom Anwender selbst kommen.

Um bei der Repräsentation der Abhängigkeiten also ein möglichst hohes Maß an Generizität zu erreichen, müssen vom Anwender die Informationen über die Abhängigkeiten bereitgestellt werden. Nur so ist es möglich, auf wechselnde Anforderungen flexibel zu reagieren.

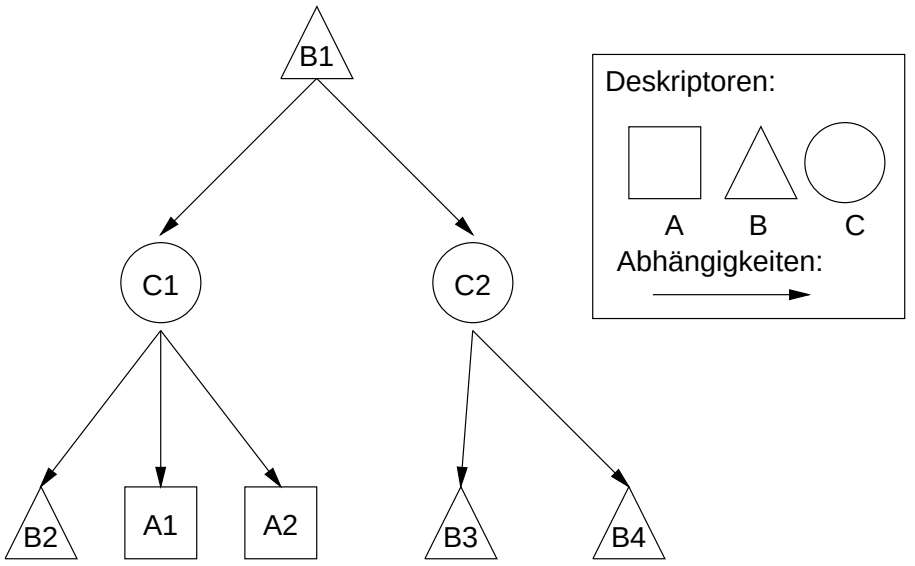

Abbildung 8.3: (Impliziter) Abhängigkeitsgraph. Darstellung der Beziehungen mehrerer Objekte mit unterschiedlichen Deskriptoren. Die Deskriptoren sind durch unterschiedliche Formen (Quadrat, Kreis, Dreieck) dargestellt, Objekte an der Objektinschrift (A1...C2) zu erkennen.

Dazu wurde der folgende Ansatz gewählt: Jedes durch das Modul *make* verwaltete Objekt ist durch einen Deskriptor gekennzeichnet. Dieser ist gewissermaßen der Typ des Objektes, wobei es sich hierbei nicht um einen Typ im Sinne von Tycoon handelt. Der Deskriptor

enthält vom Benutzer bereitgestellte Funktionen, die auf Objekten des entsprechenden Typs operieren (vgl. Schnittstelle im Anhang). Sie erfüllen folgende Aufgaben:

- Für ein Objekt, das durch diesen Deskriptor beschrieben wird, stellt die Funktion *hasChanged* fest, ob eine Transformation notwendig ist..
- Die Funktion *build* sorgt dafür, daß die für das Objekt vorgesehene Transformation ausgeführt wird.
- *dependsOn* ist eine Funktion, die für ein gegebenes Objekt alle Abhängigkeiten ermittelt. Für jedes so erhaltene Objekt muß die Funktion *dependencyFun* aufgerufen werden. Diese Funktion, die intern im Modul *make* generiert wird, steuert die weitere Systemgenerierung.

Jedem Objekt wird bei der Erzeugung der entsprechende Deskriptor zugeordnet. Es liegt in der Verantwortung des Benutzers von *make*, das Modul gemäß seiner Bestimmung zu benutzen. Natürlich ist es möglich, z. B. durch Auslassen des Aufrufs von *dependencyFun* innerhalb der eigenen *dependsOn*-Funktion für jedes abhängige Objekt ein ungewünschtes Ergebnis zu erreichen.

Bei diesem Ansatz wird also auf die explizite Modellierung oder auf den Aufbau eines Abhängigkeitsgraphen verzichtet. Die Abhängigkeitsinformation ist implizit in durch den Benutzer bereitgestellten Funktionen enthalten. Dennoch kann zugesichert werden, daß der Abhängigkeitsgraph keine Zyklen enthält. Bei Bedarf können zyklische Abhängigkeiten jedoch zugelassen werden. Sie führen dann zu keinem Abbruch der Bearbeitung. Um die Terminierung des Generierungsprozesses dennoch zu garantieren, wird kein Objekt eines Zyklus' mehr als einmal transformiert, obwohl das Aktualitätsmerkmal bereits wieder invalidiert sein kann. Hier stößt man an die Grenzen der Modellierungsmöglichkeiten dieses Ansatzes. Objekte, die durch die Generierung abhängiger Objekte selbst wieder ungültig werden, können nicht dargestellt werden.

8.3.2 Fehlerbehandlung

Während der Bearbeitung der durch den Anwender bereitgestellten Funktion können Fehler auftreten oder soll der Benutzer über den Fortgang der Bearbeitung informiert werden. Dies ist grundsätzlich nicht Sache des Moduls *make*, sondern muß von den vom Anwender bereitgestellten Funktionen *dependsOn*, *hasChanged* und *build* vorgenommen werden.

Innerhalb des Systemgenerierungsprozesses muß jedoch eine Bedingung für den Abbruch im Fehlerfall formuliert werden. Die Fehler werden durch entsprechende Meldungen dokumentiert. Es bietet sich somit an, diese Meldungen zu untersuchen und ein entsprechendes Abbruchkriterium abzuleiten. Dieser Teil muß im Modul *make* vorgenommen werden.

Dazu werden zunächst eine grobe Klassifizierung von Fehlermeldungen vorgenommen und anschließend verschiedene Möglichkeiten der Ausgabe und deren Vor- und Nachteile diskutiert.
Folgende Meldungen lassen sich unterscheiden:

Fortschrittsmeldungen: Dabei handelt es sich um Ausgaben, die zeigen, an welchem Punkt sich die Bearbeitung befindet. Sie beinhalten meist den Namen eines Objektes und die eingeleitete Operation. Beispiele sind "Loading module xxx.tm" oder "Type checking" usf.

Bearbeitungsmeldungen: Sie geben an, ob bei der eigentlichen Bearbeitung Fehler aufgetreten sind. Meldungen dieser Art lassen sich noch weiter unterteilen, etwa nach deren Schwere in fatale Fehler, Fehler, Warnungen, Informationen. Auch noch weiterreichende Einteilungen sind möglich (s. Schnittstelle zu *ErrorLog*). Für diesen Fall stehen Meldungen wie z. B. "Can't load module xxx.tm" oder "make terminated with errors".

Am Vorhandensein von Meldungen einer bestimmten Schwere kann man ablesen, daß eine weitere Bearbeitung nicht mehr sinnvoll ist. Ist z. B. bei der Übersetzung einer Schnittstelle ein Fehler aufgetaucht, so daß die Schnittstelle nicht übersetzt werden konnte, so muß die gesamte Übersetzung ebenfalls abgebrochen werden. Handelt es sich jedoch um Warnungen des Compilers, so kann es durchaus sinnvoll sein, die Bearbeitung fortzusetzen.

Das Modul *make* stützt sich bei der Ermittlung, ob Fehler aufgetreten sind, auf das Modul *errorlog*. Dementsprechend müssen die vom Benutzer bereitgestellten Funktionen Fehler, die zum Abbruch der Bearbeitung führen sollen, in einen Log schreiben und diesen als Parameter übergeben.

Diese Methode hat vor allem den Vorteil, daß Fehlermeldungen in einheitlicher Weise behandelt werden und so leicht das Auftreten eines Fehlers erkannt werden kann. Außerdem trifft man hierdurch keine Entscheidung, wie die Meldungen dem Benutzer letztendlich zugeführt werden sollen. Sowohl die Ausgabe mittels *print* als auch z. B. in einem Fenster einer grafischen Oberfläche sind denkbar.

Nachteilig ist, daß Fortschrittsmeldungen bis zum Entleeren des Logs für den Benutzer unsichtbar bleiben. Dauert beispielsweise die Abarbeitung der Funktion *build* lange und gibt währenddessen mehrere Fortschrittsmeldungen aus, so könnten diese nur gesammelt am Schluß der Funktion ausgegeben werden. Den eigentlichen Ablauf würden sie so nicht dokumentieren.

Dieser Nachteil läßt sich dadurch umgehen, daß nur die zum Abbruch führenden Meldungen in den Log geschrieben werden. Fortschritts- und andere Meldungen könnten je nach Bedarf ausgegeben werden.

Eine weitere Möglichkeit besteht darin, die Ausgabe von Fehlermeldungen vollständig der Anwenderfunktion zu überlassen und über den Rückgabewert der entsprechenden Funktion das Abbruchkriterium zu formulieren. Dies läßt sich jedoch nur auf Funktionen anwenden, die keinen weiteren Rückgabewert liefern. Auf die Funktion *hasChanged*, die für jeden Deskriptor angegeben werden muß, trifft dies z. B. nicht zu. Auch ein als Referenz übergebener Parameter löst das Problem in diesem Fall nicht, da dann keine korrekte Typisierung des ersten Parameters möglich ist.

8.4 Benutzung

Anhand eines einfachen Beispiels soll die Benutzung des Modul *make* verdeutlicht werden. Zunächst wird auf Basis eines vom Benutzer definierten Typs ein Objekt-Deskriptor erstellt. Dazu müssen die in Abschnitt 8.3.1 vorgestellten Funktionen ausprogrammiert werden. Die Signaturen können dem Anhang entnommen werden. Folgende Code-Teile können als Grundlage dienen:

```
Let Rec T <:Ok = Tuple
   ...
  var invalid :Bool
```

```
  deps :list.T(make.Object(T))
end

let hasChanged(t :T  l :errorLog.T) :Bool =
  t.invalid

let dependsOn(t :T  dependencyFun :make.DependencyFun) :Bool =
  begin
     iter.forEach(list.elements(t.deps)
       fun(object :make.Object(T))
          dependencyFun(object)
     )
     true
  end
```

Es sei an dieser Stelle darauf hingewiesen, daß die Modellierung der Abhängigkeiten durch eine Liste nur *ein* Beispiel ist. Zahlreiche andere Möglichkeiten sind denkbar. Z. B. ist vorstellbar, daß die Funktion *dependsOn* die Abhängigkeitsinformationen erst aus dem entsprechenden Objekt extrahiert. Am obigen Codefragment ist deutlich zu erkennen, daß die Behandlung der Abhängigkeiten vollständig vom Anwender vorgenommen werden muß. Dieser kann daher die zu verwendende Datenstruktur frei wählen. Die Verbindung zum Modul *Make* ist durch den Aufruf der Funktion *dependencyFun* realisiert und basiert auf der Übergabe lediglich eines Objektes. Die Reihenfolge, mit der die Objekte von *dependencyFun* bearbeitet werden, ist beliebig, solange die Abhängigkeiten zwischen den einzelnen Objekten korrekt und vollständig sind.

Durch den Aufruf von *newDescriptor* wird der neue Deskriptor instanziiert. Daraufhin können mit *newObject* beliebig viele Objekte erstellt werden, die durch diesen Deskriptor beschrieben werden. Schließlich führt

```
make.makeObject(t)
```

dazu, daß t und alle Objekte, von denen es abhängig ist, behandelt werden. Die dazu notwendige Funktion *build* ist hier nicht aufgeführt. Sie ist von dem entsprechenden Objekt bestimmt, sollte aber auf jeden Fall das Aktualitätsmerkmal (in diesem Fall *invalid*) zurücksetzen.

8.5 Bewertung

Im Gegensatz zum *Unix-Make* können bei dem hier gewählten Ansatz durch die strikte Typisierung der Sprache TL bereits zum Übersetzungszeitpunkt weitreichende Zusicherungen gegeben werden. Unter anderem wird garantiert, daß Anzahl und Typ von Funktionsparametern und -ergebnis kompatibel sind, nicht auf uninitialisierte oder undefinierte Werte, Typen oder Funktionen zugegriffen wird. Im Zusammenhang mit dem Modul *Make* bedeutet das, daß z. B. Funktionen nur auf die dafür vorgesehenen Objekte angewendet werden können. Demgegenüber kann das *Unix-Make* nicht zusichern, daß der anhand der Endung des Dateinamens oder anderweitig festgestellte Typ tatsächlich mit dem Dateiinhalt korrespondiert. Tatsächlich wird bei *Unix-Make* eher ein untypisierter Ansatz verfolgt, der es dem

Anwender und dem verwendeten Programm überläßt, festzustellen, ob die entsprechende Datei bearbeitet werden kann. Nur für bestimmte Dateiendungen werden Annahmen über die abzusetzenden Kommandos getroffen.

Der von mir vorgestellte Ansatz zwingt den Anwender, die einzelnen Funktionen der Systemgenerierung, nämlich Aktualität prüfen, Abhängigkeiten ermitteln und Übersetzen, klar voneinander zu trennen.

In bezug auf die Modulverwaltung des Tycoon Systems bietet das Modul *Make* durch sein generisches Potential weitere Möglichkeiten. Im Gegensatz zum *Unix-Make* gibt es keine Einschränkung hinsichtlich der Objekte, die behandelt werden sollen. Es können beliebige Tycoon Objekte sein. Damit ist es z. B. auch möglich, die Abhängigkeiten zwischen Moduln und Schnittstellen auf Prozedur-Ebene zu modellieren. Auf die daraus entstehenden Vorteile hinsichtlich der Einsparung von Übersetzungsvorgängen wurde bereits in Abschnitt 1.2.2 hingewiesen. Die praktische Umsetzung, die bisher noch nicht erfolgte, wird dann den tatsächlichen Nutzen zeigen. Die Abhängigkeitsinformation feiner abzustufen, wird inzwischen auch im Bereich des *Unix-Make* verfolgt [TB85, Fel88].

Einige der von der Modulverwaltung erledigten Aufgaben können mit diesem Ansatz generisch gelöst werden. Dazu gehören das Übersetzen, Laden und Importieren von Moduln, Schnittstellen und Bibliotheken. Außerdem können andere Funktionen hinzugefügt werden, wie z. B. zum Erstellen einer Repräsentation des Abhängigkeitsgraphen.

Zyklische Abhängigkeiten können sowohl verhindert als auch erlaubt und dann bei Bedarf aufgelöst werden. Es gelten allerdings die in Abschnitt 8.3.1 getroffenen Einschränkungen. Hier bestehen keine Unterschiede zum *Unix-Make*. Es erkennt ebenfalls zyklische Abhängigkeiten und löst diese automatisch auf.

Der Nachteil meines Ansatzes besteht darin, daß die meisten Aufgaben vom Anwendungsprogrammierer selbst erledigt werden müssen. Dies ist eine direkte Folge der sehr generischen Herangehensweise.

Trotz der gewonnenen Typsicherheit kann nicht garantiert werden, daß Objekte eines bestimmten Typs nur von Objekten eines anderen bestimmten Typs abhängig sein können. Entsprechende Bedingungen müssen vom Anwender selbst programmiert werden.

8.6 Ausblick

Diese Arbeit hat gezeigt, daß die existierende Modulverwaltung keine Möglichkeiten bietet, die Aufgaben der Systemgenerierung anders als in der bisherigen Weise zu lösen. Diesen Mangel zu beseitigen, ist u. a. Ziel dieser Arbeit. Die existierende Modulverwaltung galt dabei als Vorlage, die es zunächst zu verstehen galt; der überwiegende Teil der Arbeit beschäftigt sich deshalb mit diesem Thema.

Nachfolgende Arbeiten sollten den hier vorbereiteten Ansatz der generischen Modulverwaltung aufnehmen und weiterentwickeln. Dazu gehört, die bislang nur prototypisch vorhandenen Ansätze weiter auszuformulieren und auszuprogrammieren. Hinsichtlich der Aufgaben der Systemgenerierung und ganz allgemein des Konfigurationsmanagements ist zu überlegen, auch Versionen für die zu behandelnden Objekte zuzulassen. Auch die Erweiterung auf Versionsverwaltung läßt eine Generalisierung insofern zu, als daß Versionen für beliebige Objekte existieren können. Dieser Aspekt findet im Bereich der objektorientierten Datenbanken bereits Beachtung. Auf Basis von Versionen werden dort langandauernde Transaktionen modelliert.

Im Hinblick auf das Tycoon System sollte auch das bisher angewendete Modul- und Bibliothekenkonzept überdacht werden. Im Bereich der Bibliotheken haben sich die versteckten Bezeichner als wenig akzeptiert erwiesen. Geschachtelte Bibliotheken mit ebenfalls geschachtelten Namensräumen sind eine zu bedenkende Alternative und möglicherweise auf intuitivere Art zu benutzen. Bei der Revidierung dieser Strukturierungselemente sollten auch Konzepte aus anderen Sprachen betrachtet werden (Eiffel, Modula 3 usw.).

Das Tycoon System verwendet wie beschrieben sowohl den Objektspeicher als auch das Dateisystem zur Speicherung von Moduln, Schnittstellen und Bibliotheken. Zwar sprechen bislang noch einige Gründe für die Beibehaltung dieses Zustandes, es ist jedoch zu überlegen, ob nicht der Objektspeicher allein ausreichend ist. Z. B. könnten die Komponenten der Modulverwaltung ausschließlich zwischen verschiedenen Objektspeichern ausgetauscht werden.

Anhang A

Schnittstelle des Moduls *make*

```
interface Make
(*
    System:  Tycoon Compiler
    Author:  Christian Koch
    Date:    05-OCT-1995
    Purpose: generic make tool
*)

import
  errorLog :ErrorLog

export
  error :Exception end
  cycle :Exception end
  (* Raised if the dependency graph has a cycle and cycles are
     not allowed *)

  Descriptor(T <:Ok) <:Ok
  (* Describes a node in the "dependency graph". *)

  Object(T <:Ok) <:Ok
  (* object of the "dependency graph" *)

  Let DependencyFun = Fun (T <:Ok :Object(T)) :Bool
  (* 'makes' the object and returns true if it is necessary
     otherwise returns false.  This function is built
     internal. *)

  newDescriptor (T <:Ok
    built(:T  :errorLog.T) :Ok
    hasChanged(:T  :errorLog.T) :Bool
    dependsOn(:T  dependencyFun :DependencyFun) :Bool
  ) :Descriptor(T)
```

```
  (* Create a description for a node of the "dependency graph"
     from the following arguments:
     built      - Function for building a single object.
     hasChanged - Determines if the object has changed since the
                  last build.
     dependsOn  - Apply dependencyFun to all objects the object
                  depends on, raise the cycle exception if there
                  is a cycle.
  *)

  newObject (T <:Ok  obj :T  descriptor :Descriptor(T)) :Object (T)
  (* Create a new node of the "dependency graph" with the
     object obj. *)

  getObject (T <:Ok  object :Object (T)) :T
  (* Returns the 'contents' of the object. *)

  buildObject (T <:Ok  object :Object (T)) :errorLog.T
  (* Build a single object. *)

  makeObject (T <:Ok  object :Object (T)) :errorLog.T
  (* Build object and all objects it depends on. *)

  allowCycle (arg :Bool) :Ok
  (* wether the argument is false or true the build process raises
     an exeption if the "dependency graph" has a cycle or builds
     all elements of the cycle.
  *)
end;
```

Literaturverzeichnis

[AWT89] Rolf Adams, Annette Weinert, and Walter Tichy. Software change dynamics or 60 percent of all compilations are redundant. Technical Report Interner Bericht Nr. 1/89, Universität Karlsruhe, Fakultät für Informatik, Januar 1989.

[BCHT86] K. Buckner, M J. Cookson, A. I. Hinxman, and A. Tate. *Einführung in die Anwendung des UCSD p-Systems.* Vieweg, 1986.

[BV88] Frank Bomarius and Matthias Vettermann. mmf, ein *makefile*-Generator. Technical Report Interner Bericht 177/88, Universität Kaiserslautern, Fachbereich Informatik, Januar 1988.

[Car90] Luca Cardelli. The quest language and system (tracking draft). Technical report, Digital Equipment Corporation, System Research Center Palo Alto, 1990.

[Cle88] Geoffrey M. Clemm. The odin specification language. In Jürgen F. H. Winkler, editor, *International Workshop on Software Version and Configuration Control*, volume 30, pages 144–158, 1988.

[CLR86] Mario Dal Cin, Joachim Lutz, and Thomas Risse. *Programmierung in Modula-2.* B. G. Teubner, second edition, 1986.

[DuB93] Paul DuBois. *Software Portability with imake.* O'Reilly & Associates, 1993.

[Dud88] *Duden Informatik.* Bibliographisches Institut & F. A. Brockhaus AG, 1988.

[Est88] Jacky Estublier. Configuration Management: The Notion and the Tools. In Jürgen F. H. Winkler, editor, *International Workshop on Software Version and Configuration Control*, volume 30, pages 38–61, 1988.

[Fel79] Stuart I. Feldmann. Make – A Program for Maintaining Computer Programs. *Software – Practice & Experience*, 9:255–265, 1979.

[Fel88] Stuart I. Feldmann. Evolution of make. In Jürgen F. H. Winkler, editor, *International Workshop on Software Version and Configuration Control*, volume 30, pages 413–416, 1988.

[Gei83] Leo Bernhard Geissmann. *Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith.* PhD thesis, Swiss Federal Institute of Technology Zürich, 1983.

[KR90] Brian W. Kernighan and Dennis M. Richie. *Programmieren in C.* Carl Hanser Verlag, second edition, 1990.

[LS79] Hugh C. Lauer and Edwin H. Satterthwaite. The impact of mesa on system design. In *Proceedings on the 4th International Conference on Software Engineering*, pages 174–182, September 1979.

[Mat93] Florian Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer Verlag, 1993.

[Mey88] Bertrand Meyer. *Objektorientierte Softwareentwicklung.* Carl Hanser Verlag, 1988.

[MM93] Florian Matthes and Sven Müßig. The Tycoon Language TL: An Introduction. Technical Report DBIS Tycoon Report 112-93, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

[Sed88] Robert Sedgewick. *Algorithm.* Addison Wesley, second edition, 1988.

[SK88] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, 1988.

[TB85] Walter F. Tichy and M. C. Baker. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):236–244, 1985.

[Tic88] Walter F. Tichy. Tools for software configuration management. In Jürgen F. H. Winkler, editor, *International Workshop on Software Version and Configuration Control*, volume 30, pages 1–20, 1988.

[Whi91] David Whitgift. *Methods and Tools for Software Configuration Management.* John Wiley & Sons, 1991.

[Wir79] Niklaus Wirth. The module: A system structuring facility in high-level programming languages. In Jeffrey M. Tobias, editor, *Language Design and Programming Methodology*, pages 1–24. Springer Verlag, September 1979.

[Woo89] Steve Wood. *Using Turbo Pascal 5.* Programming Series. Borland Osborne/McGraw Hill, 1989.