Eberhard Karls
Universität
Tübingen

Module und Namensräume

Inhaltsverzeichnis

1 Softwaretechniksprachen

Softwaretechniksprachen wurden unter anderem mit dem Ziel entworfen, eine parallele Entwicklung von Komponenten durch unabhängige Entwickler (oder Teams oder Firmen) zu ermöglichen. Eines der Probleme, die auftreten können, wenn mehrere Entwickler gleichzeitig entwickeln, entsteht dadurch, dass Prozedur-, Variablen- oder Typnamen – allgemein: Symbole – mehrfach vergeben werden. Diese Gefahr besteht insbesondere bei „populären“ Verben und Substantiven, wie: „create“, „insert“, „get_next“, „list“ oder „handle“. Ein Namenskonflikt liegt vor, wenn bei der Erwähnung eines Namens im Quelltext nach den Regeln der verwendeten Sprache nicht mehr entschieden werden kann, welches von mehreren möglichen Konstrukten eines Programms damit referenziert wird.

Probleme, die aus einer Doppelvergabe von Namen entstehen, müssen vermieden werden. Verschiedene Sprachen stellen unterschiedliche Mechanismen bereit, um solche Konflikte auszuschließen. Wir wollen im Folgenden einen kurzen Blick auf diese Mechanismen werfen.

2 Der Übersetzungsvorgang

Rufen wir uns zuerst wieder ins Gedächtnis, dass ein typischer Übersetzungsvorgang in 2 Phasen abläuft:

1. Die Übersetzung (Compilation) jeweils einer Quelltexteinheit (meist ist das ein Modul) zu einer Objektdatei.
2. Das Zusammenbinden (Linking oder Link Loading) mehrerer Objektdateien zu einem ausführbaren Programm.

Objektdateien sind dabei fertig übersetzte Stücke ausführbaren Maschinencodes (in denen noch gewisse Löcher klaffen) zusammen mit 2 Tabellen[1]:

1. Einer Liste, an welchen Stellen noch Adressen externer Konstrukte (Prozedureinsprungspunkte, Variablen) eingefügt (gepatcht) werden müssen (Referenzenliste).
2. Einer Tabelle, welche den Namen extern verfügbar gemachter Konstrukte (Prozeduren, Variablen) die Position (Adresse) der Konstrukte in der Objektdatei zuordnet (Definitionenliste).

Der Linker fügt die Objektdateien aneinander, sammelt dabei die Adressen der enthaltenen Konstrukte (wie in der zweiten Tabelle aufgeführt), und fügt deren Adressen am Ende an den benötigten Stellen (wie in der ersten Tabelle aufgeführt) ein.

3 Namenskonflikte zur Linkzeit

Wenn die verwendete Sprache keine Mechanismen zur Vermeidung von Namenskonflikten hat, so treten diese in der Linkphase auf: Der Linker hat, wenn zwei Dateien dasselbe Symbol (sagen wir: „*foo*“) zur Verfügung stellen, und eine Objektdatei dieses Symbol benötigt, keine Möglichkeit, festzustellen, welche der beiden verschiedenen Adressen in den Objektcode der letzteren Datei eingepatcht werden sollen.

[1] Beide Tabellen kann man unter Unix übrigens mit dem Tool *nm* aus der Objektdatei extrahieren und in menschlesbarer Form ausgeben lassen. Ein Beispiel dazu findet sich im Anhang.

Ein zweites Problem, das in der Linkphase auftreten kann, ist, dass zwar an einer bestimmten Stelle eine Adresse eingefügt werden müsste, aber keine Objektdatei dieses Symbol zur Verfügung stellt. Unter Unix erhält man dann den typischen Fehler, etwa *„unresolved symbol“*.

Softwaretechniksprachen bieten vor allem Mechanismen, diese Fehler *zur Linkzeit* zu vermeiden. Motivieren lässt sich das mit einem Blick auf den typischen Ablauf eines Softwareprojektes: Beim Zusammenfügen der zugelieferten Module zu einem Ganzen (Integration) ist es bereits zu spät, noch einmal auf die einzelnen Entwickler bzw. Zulieferer zurückzugehen und Nachbesserungen einzufordern. Mechanismen zur Übersetzungszeit (Compiletime) müssen Fehler zur Linkzeit ausschließen.

4 Namensraum zur Linkzeit

Wie sehen diese Mechanismen nun bei Modula-2 aus? Aus der Perspektive des Linkers recht einfach: Wenn ein Modul – sagen wir *foo.mi* übersetzt wird, so werden alle exportierten Symbole mit dem Namen des Moduls angereichert. Wird beispielsweise die Prozedur *bar* exportiert, so wird der Einsprungspunkt in der Objektdatei *foo.o* mit *foo.bar* benannt. Da alle Module unterschiedlich heißen (müssen), und jede Objektdatei einem Modul entspricht, kann es so schon nicht mehr zu Namenskonflikten zur Linkzeit kommen: Symbole aus verschiedene Objektdateien überhaupt nicht gleich heißen.

Würde nun auch noch zusätzlich (zur Compilezeit) garantiert, daß ein Programmstück nur Namen verwenden wird (d. h. Referenzen auf diese Namen in Tabelle 1 generieren wird), die auch von einem anderen Programmstück zur Verfügung gestellt werden werden (dies ist ein Versprechen!), so könnte auch der Fehler *„unresolved symbol“* nicht mehr auftreten. In der Tat ist die Regelung dieses Versprechens genau das, was die Interface-Dateien leisten: Vor der Implementation erstellt, verkörpern sie gewissermassen die Vereinbarung, dass der Implementator eines Moduls gewisse Symbole zur Verfügung stellen wird, und impliziert die (vorweggenommene) Erlaubnis für die Entwickler der anderen Module, diese Symbole jetzt schon zu verwenden.

Damit können Probleme zur Linkzeit nicht mehr auftreten – und werden teilweise durch Probleme zur Compilezeit ersetzt, die aber zu diesem Zeitpunkt (bei der Entwicklung der Einzelmodule) viel zweckmäßiger aufgelöst werden können, als bei der Integration des Programms (Linkzeit).

5 Importe: Symbole zur Compilezeit

Wir haben bis hierher nur darauf gesehen, wie Konstrukte zur Linkzeit benannt werden: Alle Konstruktnamen wurden mit Modulnamen verziert, um Namenskonflikte zu vermeiden. Davon unabhängig ist die Frage, unter welchem Namen Konstrukte zur Compilezeit eines Moduls referenziert werden können: Der Gedanke ist zum einen der, dass man unmöglich erwarten kann, dass immer nur vollständige Linkzeitnamen eines Konstrukts (wie *foo.bar*) bei dessen Verwendung angegeben werden, zum anderen ist es auch denkbar, dass Linkzeitnamen kompliziertere Gebilde werden, als in dem vorgestellten, einfachen Schema.

Modula-2 bietet zwei Mechanismen, Namen zur Compilezeit verfügbar zu machen:

1. Den pauschalen Import aus einem Modul (hier *Foo*): `IMPORT Foo;`.
2. Den detaillierten Import einzelner Symbole (hier *bar*) aus einem Modul (hier *Foo*): `IMPORT bar from Foo;`.

Beide Importmethoden verweisen zuerst einmal auf Interface-Dateien – auf das Versprechen, dass ein anderer diese Symbole implementieren wird. Erst dadurch wird garantiert, daß kein Risiko unaufgelöster Symbole zur Linkzeit bestehen wird (sofern alle Beteiligten ihre Versprechen halten).

Der pauschale Import macht alle exportierten Konstrukte des Moduls unter ihrem voll qualifizierten Namen verfügbar – also verziert mit dem Modulnamen (etwa *Foo.bar*). Wird dieser Name verwendet, wird letzten Endes in der Objektdatei eine Referenz auf den Linkzeit-Namen des Konstrukts (hier wieder *Foo.bar*) erzeugt.

Der detaillierte Import erklärt, daß der importierte Namen in seiner Kurzform (*bar*) verwendet werden soll – aber natürlich nach wie vor dasselbe Konstrukt referenziert. Wird also nach obigem Import der Name *bar* verwendet, so wird wieder eine Referenz auf den Linkzeit-Namen *Foo.bar* erzeugt.

Die verschiedenen Importmethoden regeln also, welche Konstrukte zur Übersetzungszeit unter welchem Namen verwendet werden sollen.

6 Risiken und Nebenwirkungen

Soweit, so gut. Man fragt sich nun vielleicht, warum es in Modula keine Methode gibt, alle Konstrukte eines Moduls unter ihrem verkürzten Namen zu importieren.

Java bietet im Gegensatz zu Modula-2 genau diese Möglichkeit. Als modulare Einheit muss bei Java das *Package* gelten. Die exportierten Konstrukte sind nur Klassen – die natürlich in einer OO-Sprache sowohl für Prozeduren, Variablen als auch Typen stehen können. Mit der Deklaration

```
import wsi.tools.foo.*;
```

zu Anfang einer Klasse (sagen wir *z*) werden alle Klassen aus dem Package *wsi.tools.foo* zur Übersetzungszeit der Klasse *z* unter ihrem Kurznamen verfügbar gemacht.

Was jedoch so harmlos und praktisch aussieht, hat subtile Fallstricke, die Gegenstand der zweiten Aufgabe des zweiten Übungsblatts sind.

A Exportierte Symbole mit *nm* finden

Auf Unix lassen sich die importierten (benötigten) und exportierten (definierten und zur Verfügung gestellten) Symbole einer Objektdatei mit dem Tool *nm* einfach untersuchen. Betrachten wir den folgenden C-Quelltext, *m2.c*:

```
#include <stdio.h>

void g(){
  printf("Ich bin M2.");
}

void h(){
  printf("Ich bin M2/h.");
}


static void r(){
  printf("Ich bin M2/r.");
}
```

Nach der Übersetzung zu einer Objektdatei, *m2.o*, können wir auf diese Objektdatei das Tool *nm* anwenden:

```
bash$ nm -g m2.o

00000000 T g
00000014 T h
         U printf

bash$
```

Wir sehen, dass *g* und *h* durch das Objektmodul definiert und nach außen verfügbar gemacht werden, dagegen *printf* referenziert wird, d. h. aus einer anderen Objektdatei (oder Bibliothek) beim Linken benötigt werden wird.

Genauere Informationen sind in der Online-Hilfe (man, info) zu *nm* zu finden.