4 Sprachmechanismen zur Programmierung im Großen

In diesem Kapitel wollen wir Sprachmechanismen betrachten, die in Programmiersprachen dazu dienen, große Programme oder Programmsysteme zu erstellen. Da der Begriff „groß“ ungenau ist, betrachten wir einige *Merkmale großer Programme*:

- Das Programm wird von mehreren Personen erstellt.
- Die prozedurale Abstraktion ist zur Strukturierung alleine unzureichend, sodass weitere Strukturierungen notwendig sind.
- Unterteilung der *Daten und zugehöriger Operationen* zu eigenen Einheiten, z.B. in einem Flugreservierungssystem: Flüge, Flugzeuge, Wartelisten, ...
- Aufteilung des Systems in Module.
- Separate Programmierung dieser Module.

Mit den bisherigen Sprachmechanismen (prozedurale Abstraktion) ist dies nur unzureichend realisierbar. Notwendig für solche großen Systeme sind:

- Eine Unterteilung des Namensraumes in mehrere Bereiche.
- Die Kontrolle/Lösung von Namenskonflikten (lokal frei wählbare Namen).

Die hier vorgestellten Techniken sind weitgehend unabhängig von einem bestimmten Programmierparadigma, daher behandeln wir dies in einem eigenen Kapitel. Für konkrete Beispiele wählen wir trotzdem die imperative Programmierung. Viele Konzepte sind aber auch auf funktionale, logische oder nebenläufige Programmiersprachen übertragbar.

4.1 Module und Schnittstellen

Eine erste wichtige Technik zur Handhabung größerer Anwendungen ist die Aufteilung der Anwendung in Module. Hierbei ist ein **Modul** eine Programmeinheit mit einem Namen und folgenden Eigenschaften:

- Es ist **logisch oder funktional abgeschlossen** (also für eine bestimmte Aufgabe zuständig).
- **Datenkapselung:** Das Modul beinhaltet Daten und Operationen auf diesen Daten.

- **Datenunabhängigkeit:** Die Darstellung der Daten ist unwichtig, d. h. die Darstellung kann auch ausgetauscht werden, ohne dass dies Konsequenzen für den Benutzer des Moduls hat (außer, dass die implementierten Operationen eventuell schneller oder langsamer ablaufen).
- **Informationsverbergung** (information hiding): Die Implementierung dieser Daten und Operationen ist außerhalb des Moduls nicht bekannt.

Somit entspricht ein Modul in etwa einem ADT, wobei in imperativen Programmiersprachen das Modul noch einen lokalen Zustand enthalten kann.

Somit hat ein Modul zwei Sichten:

1. **Implementierungssicht:** Wie sind die Daten/Operationen realisiert?
2. **Anwendersicht:** Welche (abstrakten) Daten/Operationen stellt das Modul zur Verfügung? Dies wird beschrieben durch die **Schnittstelle** des Moduls.

Anmerkungen:

- Die Anwender eines Moduls sollen nur die Schnittstelle kennen, d. h. sie dürfen nur die Namen aus der Schnittstelle verwenden ($\rightsquigarrow$ information hiding).
- Ein Modul kann durchaus mehrere Schnittstellen haben (z. B. in Modula-3, Java).

Beispiel 4.1 (Modul `Table`).

```
procedure insert (x: string; v: int);
function lookup (x: string): int;
```

} Schnittstelle

```
var maxentry = 1000;
var last = 0;
var table: array [1000] of ...
procedure insert (x: string, v: int) is
   begin ... end
function lookup (x: string): int is
   begin ... end
```

} Implementierung

Module unterteilen den Namensraum in

- sichtbare Namen (hier: `insert`, `lookup`) und
- verborgene Namen (hier: `maxentry`, `last`, `table`).

Ein **Modulsystem**

- prüft den korrekten Zugriff auf Namen,
- vermeidet Namenskonflikte (z.B. sollten identische lokale Namen in verschiedenen Modulen keine Probleme bereiten),

- unterstützt häufig die **getrennte Übersetzung** von Modulen:
  - Übersetzung und Prüfung eines Moduls, auch wenn nur die Schnittstellen (und nicht die Implementierung) der benutzten anderen Module bekannt sind. Hierdurch wird insbesondere eine unabhängige Softwareentwicklung unterstützt.
  - In diesem Fall erfolgt die Erzeugung eines ausführbaren Gesamtprogramms durch einen Binder (Linker).
- unterstützt **Bibliotheken** oder **Pakete**: Zusammenfassung mehrere Module zu größeren Einheiten (z.B. Numerikpaket, Fensterbibliothek, Graphikpakete, Netzwerkpakete,...).

Beispiel 4.2 (Module in Modula-2).
Schnittstellendefinition:

```
DEFINITION MODULE Table;
  (* Liste aller sichtbaren Bezeichner *)
  EXPORT QUALIFIED insert, lookup;
  PROCEDURE insert (x: String; v: INTEGER);
  FUNCTION lookup (x: String): INTEGER;
END Table.
```

Implementierung:

```
IMPLEMENTATION MODULE Table;
  VAR maxentry, last: INTEGER;
  VAR table: ARRAY [1..maxentry] OF ...;

  PROCEDURE insert (x: String; v: INTEGER);
  BEGIN
    ⋮ (* Implementierungscode *)
  END insert;

  FUNCTION lookup (x: String): INTEGER;
  BEGIN
    ⋮ (* Implementierungscode *)
  END lookup;
END Table.
```

Benutzung (Import) eines Moduls:

```
IMPORT Table;
```

oder

```
FROM Table IMPORT insert, lookup;
```

Dadurch sind von nun an die Namen `Table.insert` und `Table.lookup` sichtbar und können wie lokale definierte Prozeduren verwendet werden.

Die **modulare Programmierung** ist also durch folgendes Vorgehen charakterisiert:

- Ein Modul entspricht in etwa einem ADT mit einem internen Zustand.
- Ein Programm entspricht einer Menge von Modulen.
- In den Schnittstellen wird nur die zur Benutzung notwendigen Bezeichner bekanntgegeben.
- Bei der Ausführung eines Gesamtsystem wird der Rumpf des **Hauptmoduls** ausgeführt.