Kapitel 5: Methodisches Programmieren

Beispiel einer komplexen Anwendung

In einer Firma sollen durch ein Programm Quittungen erstellt werden:

Fa. L. Kaufgut *12.1.2004*

Ladenstr. 17

77777 Handelsstadt

QUITTUNG

Wir bestätigen, von Herrn P. Schulze

EUR 218,37 (zweihundertachtzehn)

erhalten zu haben.

Diese Quittung wurde maschinell erstellt und trägt keine Unterschriften.

Beispiel einer komplexen Anwendung

Aufgabe: Bei Eingabe von Datum, Kundenname und Betrag soll eine Quittung in dieser Form erstellt werden.

Modellierung:

Kundenname:	`string`
Quittungstext:	`string`
Datum:	`string` oder `datum_t` (benutzerdefiniert)
Betrag:	`float`

Grobgliederung in Teilaufgaben

- Erzeugung der Wortdarstellung zum Zahlungsbetrag,
- Einsetzen der Eingaben und der erzeugten Wortdarstellung in den festen Quittungstext,
- Erstellung der endgültigen Fassung gemäß gewünschtem Layout. Evtl. Zeilenumbruch erforderlich.
- Eingabe der Parameter, Drucken der Quittung.

Modulkonzept

Ein *Modul* ist eine Menge von Datentypen zusammen mit einer Menge von Funktionen auf diesen (und anderen) Typen und evtl. noch weiteren Bestandteilen.

Rechenstrukturen sind insbesondere Module.

Im allgemeinen enthält ein Modul interne Bestandteile, die von außen nicht verwendbar sein sollen. Diejenigen Bestandteile, die von anderen verwendbar sind, heißen *Schnittstelle*.

Modularisierung bezeichnet die Zerlegung einer Aufgabe in Teile und Festlegung der für die einzelnen Teile zu erstellenden Schnittstellen.

Modularisierung im Beispiel

Module für die Datentypen `string` und `float`, dazu:

`Wortdarst`	Erzeugung der Wortdarstellung zum Betrag
`Trennen`	Korrekte Trennung von Wortdarstellungen
`Quittung`	Einsetzen der Eingaben, Erstellung der endg. Quittung

Schnittstelle von `Wortdarst`

konvert $=$ **function**(x:**float**) :

pre $x \geq 0$

result *konvert*(x) $=$ Wortdarstellung von x

Centanteile werden nicht berücksichtigt

Verwendung eines Moduls

Die Schnittstelle definiert die nach außen sichtbaren Datentypen und Funktionen mit ihren Typen.

Die Implementierung der Funktionen wird nicht sichtbar gemacht.

Von außen benutzt man die Funktionen allein aufgrund ihrer Spezifikation.

Spezifikation = Informelle oder formale Beschreibung der Funktion.

Vorteile der Modularisierung

- Strukturierung großer Programmsysteme
- Abkapselung: Implementierung des Moduls von dessen Verwendung getrennt.
- Verstecken von Hilfsfunktionen: Nur explizit in der Schnittstelle aufgeführte Funktionen und Typen sind von außen zu verwenden. Vermeidung von Namenskonflikten
- Änderungen überschaubar durchführbar
- Einzelimplementierungen unabhängig voneinander.

Arten der Modularisierung

Es gibt unterschiedliche Leitlinien für die Abgrenzung von Modulen:

- Problemorientierung: Zusammenfassung von Algorithmen die ein bestimmtes Teilproblem lösen.
- Datenorientierung: Zusammenfassung geeigneter Datentypen und zugehörige Funktionen.
- Funktionsorientierung: Zusammenfassung funktional verwandter Algorithmen, z.B.: “Statistikbibliothek”.

Objektorientierung (2. Semester) versucht, diese Arten der Modularisierung miteinander zu verbinden.

Programmentwicklung

Die Modulstruktur liegt nur selten von vornherein statisch vor. Stattdessen:

- Weitere Aufteilung eines Moduls in Teilmodule.
- Zusammenfassung bisher getrennter Module in eins, z.B.: `Trennen` in `Wortdarst` integrieren.
- Erweitern von Schnittstellen
- Veränderung der Spezifikation von Schnittstellen

Modularisierung in OCAML

Schnittstellen werden in OCAML durch *Signaturdeklarationen* festeglet,

Module werden in *Strukturdeklarationen* beschrieben.

Beispiel:

```
module type WortdarstSig = sig
val konvert : float -> string
end

module Wortdarst : WortdarstSig = struct
let pi = 3.141
let konvert x = "*** Rechnungsbetrag ***" (* muss verbessert werde
end
```

Zugriff auf Module

Jedes Modul hat eine Signatur; ist keine explizit angegeben, so wird eine solche automatisch erstellt.

Auf die Komponente x des Moduls M wird mit $M.x$ zugegriffen, z.B.: ist

```
Wortdarst.konvert : float -> string
```

Beachte: `Wortdarst.pi` ist nicht definiert.

Durch die Deklaration `open` M wird der Modul M geöffnet, man kann dann die Bestandteile der Schnittstelle direkt verwenden. Die "versteckten" Bestandteile, wie `pi` aber nicht.

Vordefinierte Module

Im OCAML System sind zahlreiche Module bereits vordefiniert, so z.B.:
`List`, `String`, `Array`, `Graphics`, ...