D, eine Weiterentwicklung der Programmiersprache C

ABC-Schütze

C zählt trotz bekannter Probleme zu den populärsten Programmiersprachen. Eine der jüngeren Bewerberinnen um die Nachfolge ist D: Objektorientiert, inklusive Garbage Collection und Unit-Test-Funktionalität schleppt diese Sprache weit weniger Ballast mit sich herum als etwa C++. Friedrich Dominicus

Derzeit vorherrschende Sprachen wie C, C++, Java, Perl oder Python sind imperativ-objektorientiert. In diese Kategorie fällt auch die etwas exotische Sprache D [1]. Der Name deutet es an: D ist als Nachfolgerin von C konzipiert, sie soll mit deren Unzulänglichkeiten aufräumen und dabei effizienter und leichter zu erlernen sein als C++. Walter Bright – Autor von D – charakterisiert seine Entwicklung als System- und Applikationssprache, die zwar auf einem höheren Level als C++ angesiedelt ist, sich aber auch für Low-Level-Aufgaben wie Kernelprogrammierung eignet.

Stärken und Schwächen

Eine große Schwäche von D ist die mangelnde Unterstützung durch Werkzeuge. Die Standardbibliothek ist – selbst nach Aussage des Autors – arg unterentwickelt. Für die GUI-Entwicklung auf Linux existiert lediglich eine Alphaversion des GTK-Bindings DUI ([3] und Abbildung 1). Debugger und Code-Browser sind zumindest für Linux nicht vorhanden. Diese Vernachlässigung teilt D mit einer Reihe anderer Sprachen. Dass es auch anders geht, beweisen Implementierungen für Smalltalk, Common Lisp, Scheme, Erlang oder Beta.

Einfach zu lernen ist D vor allem für C- und C++-Programmierer. Auf den ersten Blick sieht ein D-Programm wie C aus (siehe Listing 1). Kompilieren lässt sich das Beispiel mit dem DMD-Compiler [2]: »dmd greetings.d«. Der Aufruf »./greetings Hallo Leser« führt zu folgender Ausgabe:

```
args[0] = './greetings'
args[1] = 'Hallo'
args[2] = 'Leser'
```

Näher betrachtet zeigt jedoch schon die Deklaration der »main()«-Funktion, dass es sich nicht um C handelt. Die Notation »[]« steht für ein dynamisches Feld, in dem implizit die Länge vorliegt. Ähnliches ist zwar auch in C möglich, dort aber nur für eindimensionale Felder.

Zu den wichtigen Neuerungen von D gegenüber C zählen Klassen, Schnittstellenbeschreibungen (Interfaces), Schablonen (Templates), die strukturierte Fehlerbehandlung (Exceptions) und die automatische Speicherverwaltung (Garbage Collection). Von Eiffel entlehnt D das Design by Contract. Überladen von Funktionen ist gestattet. Ähnlich wie Java (aber im Gegensatz zu C++) kennt das Klassenkonzept von D zwar nur Einfachvererbung, mit Interfaces lässt sich die Mehrfachvererbung aber nachbilden. Das Beispiel in Listing 2 verwendet Interfaces, Klassen, Vererbung und teilweise Design by Contract.

Klassendeklaration

Der Code in Zeile 9 deklariert eine Klasse, die zwei Interfaces implementiert (Zeilen 1 und 5). Wenn eine Klasse von einer Schnittstelle abgeleitet wird, muss sie alle in der Schnittstelle vorliegenden Funktionen implementieren. Java-Programmierer kennen dieses Konzept, auch ihre Sprache nutzt Interfaces. Das Beispiel leitet weitere geometrischen Figuren von »Figure« ab (Circle, Rectangle und Triangle).

Die Implementierung der Klasse »Rectangle« (Zeile 15 bis 48) zeigt einige wichtige Sprachkonzepte. Die Klassendeklaration ist der von C++ sehr ähnlich. Nach dem Schlüsselwort »class« folgen der Klassenname, ein Doppelpunkt »:«, die Superklasse sowie eine oder mehrere Schnittstellenbeschreibungen (Zeile 15). Die »import«-Anweisung (Zeile 16) lädt zusätzliche Module. Diese spannen einen eigenen Namensraum auf und vermeiden so Probleme, die bei einem globalen Namensraum auftreten. Namenskollisionen lassen sich leicht auflösen, indem man der Funktion den Modulnamen voranstellt.

D unterscheidet bei den Zugriffsrechten zwischen »private«, »protected«, »public« und »export«. Als »private« deklarierte Elemente sind nur innerhalb der Klasse sichtbar. Für »private«-importierte Module gilt diese Sichtbarkeitsregel ebenfalls, ihre Funktionen werden

Listing 1: Ein erstes D-Programm

```
01 int main(char[][] args) {
02    for (int i = 0; i < args.length; i++)
03       printf("args[%d] = '%s'\n", i, (char *)args[i]);
04    return 0;
05 }
```

kein Bestandteil der Schnittstelle der sie importierenden Klasse. Die Zeilen 19 und 20 deklarieren zwei private Felder der Klasse, hier die beiden Seiten eines Rechtecks. Auf exportierte Symbole (Schlüsselwort »export«) können auch Programme außerhalb des entwickelten Systems zugreifen. Nützlich ist das vor allem beim Erstellen von dynamischen Bibliotheken.

Neue Objekte lassen sich mit »new *Klassenname*« erzeugen. Standardmäßig verwendet D Garbage Collection, Objekte explizit freigeben ist daher nicht nötig. Die Konstruktoren tragen den Namen »this« (Zeilen 26 und 31). Da das Überladen von Funktionen erlaubt ist, kann eine Klasse viele Konstruktoren erhalten, die sich im Typ ihrer Argumente unterscheiden.

Verträge

Die »area()«-Funktion (Zeilen 36 bis 42) benutzt eine hilfreiche Erweiterung, die so genannten Verträge. Sie legen offen, welchen Anforderungen übergebene Parameter genügen müssen (»in«-Blöcke) und was nach der Ausführung einer Methode gilt (»out«-Blöcke). Im Beispiel gibt es keine Vorbedingung, der Rückgabewert muss aber dem Produkt der beiden Seiten entsprechen. Da die Methode mit Fließkommazahlen rechnet, empfiehlt es sich, nur mit begrenzter Genauigkeit zu testen: »feq()« ist einem »= =«-Vergleich vorzuziehen.

Das »assert()« ähnelt dem von C, allerdings moniert es bei einer Verletzung

Listing 2: Objektorientiertes D-Programm

```
001 interface InterfaceEx1 {
002    double area ();
003 }
004
005 interface InterfaceEx2 {
006    double perimeter();
007 }
008
009 class Figure : InterfaceEx1, InterfaceEx2 {
010    /* Dummy-Klasse */
011    double area () { return 0.0;}
012    double perimeter (){ return 0.0;}
013 }
014
015 class Rectangle : Figure {
016    private import math2;
017
018    private:
019    double side_1;
020    double side_2;
021
022    export {
023       double get_side_1() {return side_1;}
024       double get_side_2() {return side_2;}
025
026       this () {
027          side_1 = 1.0;
028          side_2 = 1.0;
029       }
030
031       this (double s1, double s2){
032          side_1 = s1;
033          side_2 = s2;
034       }
035
036       double area ()
037          out (result) {
038             assert(feq(result, (get_side_1() *
    get_side_2())));
039          }
040          body {
041             return (get_side_1()  *
    get_side_2());
042          }
043
044       double perimeter () {
045          return (2 * (get_side_1() +
    get_side_2()));
046       }
047    }
048 }
049
050 class Circle: Figure {
051    import math;
052
053    private:
054    double radius;
055
056    public {
057       this () { radius = 1.0; }
058       this (double init_val) { radius =
    init_val; }
059       double get_radius () { return radius; }
060       double area () { return (PI * radius *
    radius); }
061       double perimeter () { return (PI * 2 *
    radius); }
062    }
063 }
064
065 class Triangle : Figure {
066    private import math2;
067    private import math;
068
069    private:
070    double epsilon = 0.001;
071    double side_a, side_b, side_c;
072
073    public {
074       this () {
075          side_a = side_b = side_c = 1.0;
076       }
077
078       this (double sa, double sb, double ag) {
079          /* Given side a, side b and the
    enclosing angle */
080          double angle_gamma = deg2rad(ag);
081          side_a = sa; side_b = sb;
082          side_c = sqrt((sa * sa) + (sb * sb) -
083             2 * side_a * side_b *
    cos(angle_gamma));
084       }
085
086       double area () {
087          /* Heron'sche Formel */
088          double s = ((side_a + side_b +
    side_c) / 2);
089          return (sqrt(s * (s - side_a) * (s -
    side_b) * (s - side_c)));
090       }
091
092       double perimeter () {
093          return side_a + side_b + side_c;
094       }
095    }
096 }
097
098 unittest {
099     assert(10.001 ==  10.00);
100     assert(feq (rad2deg(deg2rad (90.0)),
    90.0));
101 }
102
103 import c.stdio; // für sscanf
104 int main (char[][] args){
105    char [] v1;
106    char [] v2;
107    double dv1, dv2;
108
109    sscanf(args[1], "%lf", &dv1);
110    sscanf(args[2], "%lf", &dv2);
111
112    Figure fig = new Rectangle(dv1, dv2);
113    printf("Rectangle: area = %.2lf, perimeter
    = %.2lf\n",
114       fig.area(), fig.perimeter());
115
116    fig = new Circle(10.0);
117    printf("Circle area = %.2lf, perimeter =
    %.2lf\n",
118       fig.area(), fig.perimeter());
119
120    fig = new Triangle();
121    printf("Triangle  area = %.2lf, perimeter =
    %.2lf\n",
122       fig.area(), fig.perimeter());
123
124    return 0;
125 }
```

des Kontrakts eine Ausnahme (Exception). Behandelt das Programm die Ausnahme nicht, sorgt D für eine Meldung, in der steht, an welcher Stelle das Programm einen Vertrag gebrochen hat. Gerade während der Entwicklung ist dieses Verhalten praktisch.

In den Zeile 112 bis 122 zeigt sich der Vorteil der Basisklasse »Figure«: Der Code weist der Variablen »fig« nacheinander Objekte der Klassen »Rectangle«, »Circle« und »Triangle« zu. Obwohl »fig« vom Typ »Figure« ist, ruft »fig.area()« immer die Implementierung auf, die zum jeweiligen Objekt gehört. Allgemeiner ausgedrückt: In D kann der Programmierer einem Vorgängerobjekt Nachfolgerobjekte zuordnen (Covariance). Implementiert ein Nachfolger genau die Schnittstelle des Vorgängers, dann lassen sich Funktionen durch das Vorgängerobjekt aufrufen. Tatsächlich ausgeführt wird aber die Implementierung des Nachfolgers.

Diese kovariante Zuordnung ist speziell in größeren Programmbibliotheken recht nützlich, da sich damit leicht Schablonen für Algorithmen erstellen lassen. So ist die Ausgabe einer Figur, die aus verschiedenen Elementen (Kreisen, Polygonen …) besteht, recht einfach – vorausgesetzt jedes Element implementiert eine Methode wie »draw()«.

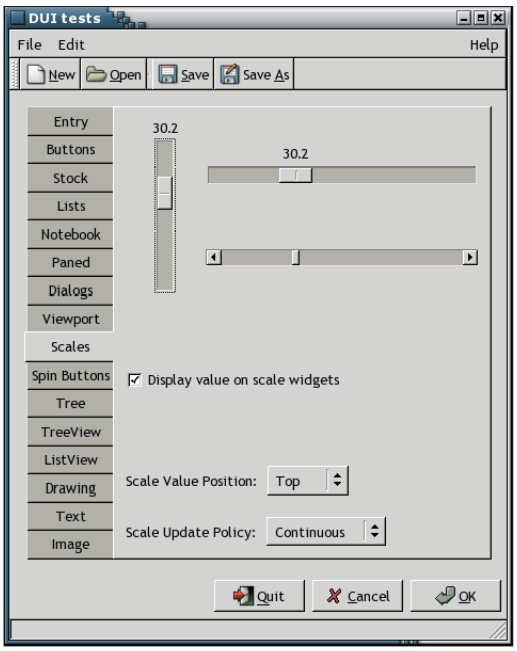

Abbildung 1: DUI gibt D-Programmierern Zugriff auf die GTK-Widgetbibliothek. Allerdings ist DUI erst im Alphastadium, mehr als solche Testprogramme sind derzeit kaum zu erwarten.

In D sind übrigens alle Objekte Referenzen (Pointer auf Klassenstrukturen). Dennoch folgt die Aufrufsyntax für Klassenmethoden und Attribute der einfachen »*Objektname.feature()*«-Konvention; C- und C++-Programmierer würden hier eher »->« erwarten.

Templates und Ausnahmen

Schablonen (Templates) sind ein zentrales Element für die wieder verwendbaren Komponenten in statisch getypten Sprachen. Auf dem FTP-Server des Linux-Magazins [4] ist ein ausführliches Beispiel zu finden. Folgender Code definiert eine Vorlage »LinkedListTemplate«, die einen generischen Parameter »T« akzeptiert:

```
template LinkedListTemplate(T) {
   class LinkedList {
      T item;
      LinkedList next;
      uint count;
      private LinkedList head;
      private LinkedList last;
```

Innerhalb der Schablone ist eine Klasse definiert, die sich auf diesen generischen Typ bezieht. Um eine Instanz dieser Klasse zu erzeugen, die mit dem gewünschten Typ versehen ist, ist etwas Tipparbeit nötig, die sich aber mit dem »alias«-Schlüsselwort umgehen lässt. In Listing 3 sorgt Zeile 2 dafür, dass »IntList« als Ersatz für die verschachtelte Klassendefinition dient: »IntList mylist« erzeugt eine verkettete Liste, die Elemente vom Typ »int« aufnimmt.

Eines der bemerkenswerten Features von D sind die integrierten Unit-Tests, die mit dem Schlüsselwort »unittest« beginnen. In Listing 3 ist ein solcher Testblock zu sehen; er kann an beliebiger Stelle im Code stehen. Damit lassen sich Code und Test hervorragend integrieren.

Eine weitere Technik, die für mehr Codequalität sowie robustere Programme sorgt, sind Ausnahmen (Exceptions). Ausnahmen sind in D immer Instanzen von Klassen, die von »Exception« oder »Error« erben sollten. Um einen Fehler anzuzeigen, wirft »throw« ein Ausnahmeobjekt. Tritt innerhalb eines »try«-Blocks eine Ausnahme auf, dann kann eine passende »catch(*Ausnahme*)«-Anweisungen die Exception fangen und geeignet reagieren.

Fazit

D ist eine interessante Weiterentwicklung von C und trägt trotz Klassen, Templates, Ausnahmen und Überladen von Funktionen weitaus weniger Ballast als C++ mit sich herum. Wer sich in C und objektorientierter Programmierung auskennt, hat auch D schnell gelernt. An einigen Stellen ist der Neuling jedoch unelegant, zum Beispiel bei Templates: Es wäre einfacher, eine Klasse mit generischen Parametern zu verwenden, etwa nach dem Vorbild von Eiffel.

Leider sind Teile der in der Dokumentation beschriebenen Funktionalität noch nicht implementiert. Die größten Nachteile hat D aber im Bereich der Werkzeugunterstützung. Trotz dieser Defizite sollten C-Programmierer einen Blick auf D werfen: Die Kombination aus Erweiterungen, die zu robuster und gut dokumentierte Software führen sollen, mit wichtigen Funktionen für die systemnahe Programmierung macht D zu einer interessanten Sprache. *(fjl)* ■

Infos

[1] D-Homepage und Handbuch: [http://www.digitalmars.com/d/]
[2] DMD-Compiler: [http://www.digitalmars.com/d/dcompiler.html]
[3] DUI: [http://ca.geocities.com/duitoolkit/]
[4] Listings: [ftp://ftp.linux-magazin.de/pub/listings/magazin/2004/01/D/]

Listing 3: Unit-Test

```
01 unittest {
02    alias instance LinkedListTemplate(int).LinkedList
   IntList;
03    IntList res_list = new IntList();
04    res_list.insert_last(1);
05    res_list.insert_last(2);
06    res_list.insert_last(3);
07    assert(res_list.item_at(0) == 1);
08    assert(res_list.item_at(1) == 2);
09    assert(res_list.item_at(2) == 3);
10 }
```