- Sie können *Kommandos für die Haltepunkte* angeben, die dann anstelle eines Stopps ausgeführt werden.
- Sie haben nicht nur Lesezugriff auf alle Programmteile, sondern können gleichzeitig *Variablen verändern* und von der Kommandozeile aus *Funktionen aufrufen*.
- Zusätzlich zu den Haltepunkten gibt es so genannte *Watchpoints*; mit deren Hilfe wird das Programm dann gestoppt (oder ein anderer Befehl ausgeführt), wenn sich der Wert einer Variablen, eines Attributs oder eines Ausdrucks davon verändert. Ähnlich arbeiten die *Catchpoints*, die unter anderem dann eine Aktion veranlassen, wenn eine C++-Ausnahme geworfen wird (siehe Seite 366). Natürlich lassen sich auch Prozesssignale abfangen.
- Der *gdb* kann auch *parallelisierte Programme* untersuchen, die mit mehreren Threads oder Prozessen arbeiten.
- Der *gdb* ist in der Lage, *bereits laufende Programme* zu debuggen. Dazu geben Sie beim Aufruf hinter dem Programmnamen einfach die Prozessnummer an. Das kann für die Fehlersuche bei Multitier-Anwendungen sehr hilfreich sein. Darüber hinaus können Sie sogar Programme verfolgen, die *auf einem anderen Rechner* (oder einem angeschlossenen Echtzeitsystem) laufen.

Sein großer Leistungsumfang, seine Zuverlässigkeit und Flexibilität haben den *gdb* zum Standard-Debugger unter Linux gemacht. Fast alle anderen Debugging-Werkzeuge sind lediglich Frontends, das heißt gekapselte Benutzerschnittstellen zum *gdb*. Er ist jedoch nicht auf Linux beschränkt, sondern auf nahezu sämtlichen Unix-Plattformen ebenso verfügbar.

6.2.5 Der grafische Debugger *DDD*

So leistungsfähig der *gdb* auch ist – seine Bedienung über die Kommandozeile setzt jedoch genaue Kenntnisse über die Befehle und ihre Argumente voraus. Vielen PC-Umsteigern mutet eine solche Arbeitsweise zudem sehr spartanisch an. Für all jene ist der grafische *Data Display Debugger* DDD zu empfehlen.

Ursprünge und Bezugsquelle

Er ist entstanden aus Diplom- und Doktorarbeiten von Dorothea Lütkehaus und Andreas Zeller an der Technischen Universität Braunschweig. Obwohl die Autoren diese Hochschule mittlerweile verlassen haben, wird er dort immer noch gepflegt und archiviert. Da er ebenfalls unter der GNU General Public License (GPL, siehe Seite 9) steht, sind viele Entwickler weltweit an seiner Weiterentwicklung beteiligt.

Sie können stets die neueste Version über die Webseite des GNU-Projekts *www.gnu.org/software/ddd* beziehen. Aber auch bei den meisten Linux-Distributionen ist der *DDD* mittlerweile enthalten.

Andere gdb-*Frontends*

Der Data Display Debugger ist keineswegs die einzige grafische Benutzerschnittstelle zum *gdb*; neben den eigenständigen Frontends *xxgdb* und `tgdb` bringen auch die meisten integrierten Entwicklungsumgebungen eigene Aufsätze auf den *gdb* mit, beispielsweise Kdbg in *KDevelop* (siehe Seite 506) oder der SNiFF-Debugger in *SNiFF+* (Seite 536). Aufgabe jedes dieser Programme ist es, dem Benutzer die Eingabe der *gdb*-Befehle auf der Kommandozeile zu ersparen und die Bedienung vorwiegend auf Mausklicks zu beschränken; dabei legt man natürlich besonderen Wert auf die gebräuchlichsten Befehle und lässt seltenere außen vor.

Die grafische Datenanzeige

Die Besonderheit am *DDD*, der er auch seine Popularität verdankt, ist seine Fähigkeit, komplexe Datenstrukturen als Graphen zu visualisieren. Durch einfache Mausklicks kann der Benutzer Zeiger dereferenzieren oder die Inhalte von Objekten darstellen lassen. Besonders bei verschachtelten Objekten lassen sich die Zusammenhänge sehr gut durch automatisch verwaltete Bäume veranschaulichen. Auf Wunsch dringt die Anzeige dabei immer tiefer in die Verschachtelungen vor.

In diesem Abschnitt setze ich voraus, dass Sie die Grundbegriffe des Debuggens und des Einsatzes des *gdb* kennen, wie ich sie im letzten Abschnitt ab Seite 440 beschrieben habe. Zunächst wollen wir uns ansehen, wie die *gdb*-Kommandos über den *DDD* erreichbar sind. Anschließend werden wir einen Blick auf die Visualisierung von Datenstrukturen werfen.

Elementare Bedienung

Start des DDD *und seine Oberfläche*

Die Art des Aufrufes ist genau dieselbe wie beim *gdb*; Sie müssen lediglich die ersten zwei Buchstaben austauschen. Als Argument können Sie den Namen des Programms und eventuell eine Speicherauszugsdatei oder eine Prozessnummer angeben. Für unser Beispiel heißt das:

```
% ddd DebugTest
```

Die Fenster der Oberfläche

Gleich anschließend sehen Sie das DDD-Fenster vor sich (Abbildung 6.1). Die Aufteilung des Fensters dürfte klar sein: Menüleiste, Werkzeugleiste, Quelltextfenster, Debugger-Konsole und Statusleiste. Je nach Situation können noch weitere Teile hinzukommen, zum Beispiel das grafische Datenfenster. Wenn Sie möchten, stellt Ihnen der *DDD* auch alle Teile in separaten Fenstern dar (siehe EDIT | PREFERENCES | STARTUP | WINDOW LAYOUT).

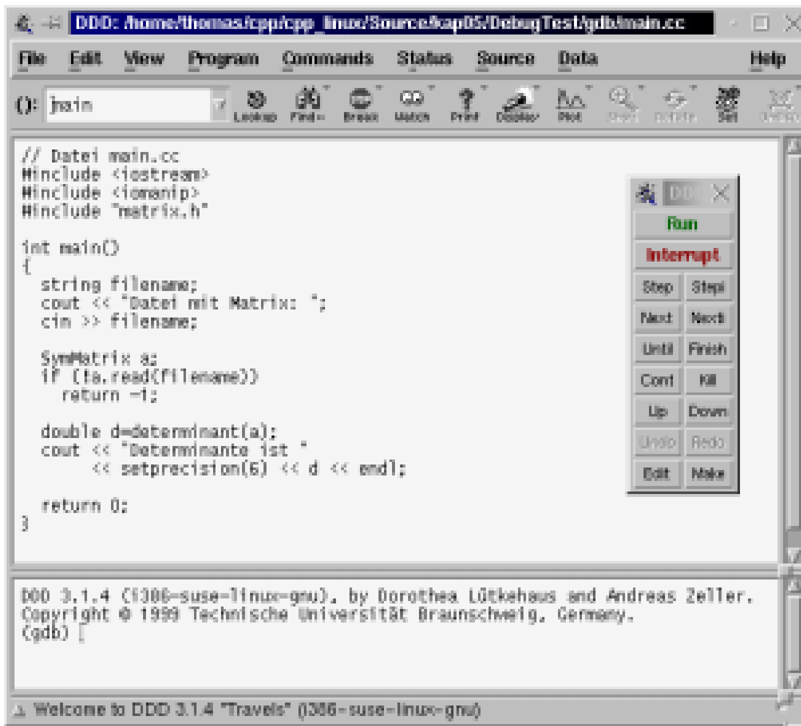


Abbildung 6.1
Die Benutzeroberfläche des DDD bietet einfachen Zugriff auf alle häufig benötigten Funktionen.

Die Kommandoleiste

Besonders praktisch ist das separate Werkzeugfenster für die wichtigsten Kommandos (in Abbildung 6.1 über der rechten Hälfte des Quelltextfensters). Darüber können Sie die Einzelschrittausführung bequem steuern. Im Folgenden wollen wir dieses Fenster kurz als *Kommandoleiste* bezeichnen. (Über EDIT | PREFERENCES | SOURCE | TOOL BUTTONS LOCATION lassen sich die enthaltenen Schaltflächen aber auch im Quelltextfenster platzieren. Überhaupt können Sie mit dem PREFERENCES-Dialog den *DDD* sehr weitreichend an Ihre persönlichen Vorlieben anpassen.)

Start eines Programms

Am einfachsten starten Sie Ihr Programm über die Schaltfläche RUN der Kommandoleiste oder mit der Taste F2. Wenn Sie Kommandozeilenargumente brauchen, können Sie diese über den Dialog festlegen, den Sie mittels des Menüpunktes PROGRAM | RUN erhalten. Der *DDD* speichert dort sogar mehrere Folgen von Argumenten, aus denen Sie sich eine aussuchen können.

Setzen von Haltepunkten

Auch für die Festlegung von Haltepunkten gibt es mehrere Möglichkeiten. Am einfachsten ist der Doppelklick am Anfang der Codezeile. Zur Kennzeichnung des Haltepunktes erscheint ein Stoppschild. Alternativ gibt es den Punkt SET BREAKPOINT aus dem Kontextme-

nü (öffnet sich durch Klick auf die rechte Maustaste) oder dem Stopp-Symbol aus der Werkzeugleiste. Wenn Sie Haltepunkte über Funktionsnamen festlegen wollen oder sonstige komplexere Vorgaben beabsichtigen, ist der Menüpunkt SOURCE | EDIT BREAKPOINTS das Richtige. Schließlich steht es Ihnen auch frei, das Kommando `break` in die *gdb*-Konsole einzugeben.

Ausgabe von Programmcode

Standardmäßig lädt *DDD* die Quelltextdatei, die die `main()`-Funktion enthält. Alle weiteren beteiligten Dateien können Sie über FILE | OPEN SOURCE auswählen und laden.

Einzelschritte und Fortsetzung

Ein Programm in einzelnen Schritten unter Beobachtung ablaufen zu lassen, ist beim *DDD* überaus einfach. Starten Sie das Programm über RUN auf der Kommandoleiste und lassen Sie es bis zu einem eingestellten Haltepunkt laufen. Dann stehen wieder die Befehle `step`, `next`, `until` und `finish` zur Verfügung (siehe Seite 446) – alles über die gleichnamigen Schaltflächen auf der Kommandoleiste. Auch der Befehl `cont` zum Fortsetzen ist darüber erreichbar.

Verfolgung der Aufrufkette

Wenn Sie innerhalb einer Funktion zum Halten kommen, können Sie sich alle Funktionen ausgeben lassen, über die das Programm hierher gelangt ist. Was beim *gdb* noch `backtrace` heißt, findet sich hier unter dem Menüpunkt STATUS | BACKTRACE. Die Darstellung erfolgt über ein Dialogfenster (Abbildung 6.2), das noch eine weitere Hilfestellung in sich birgt. Mittels der Schaltflächen UP und DOWN können Sie auch im Quelltextfenster zu den jeweiligen Programmstellen springen, an denen der Aufruf steht. Dort können Sie dann nach Wunsch weitere Untersuchungen anstellen.

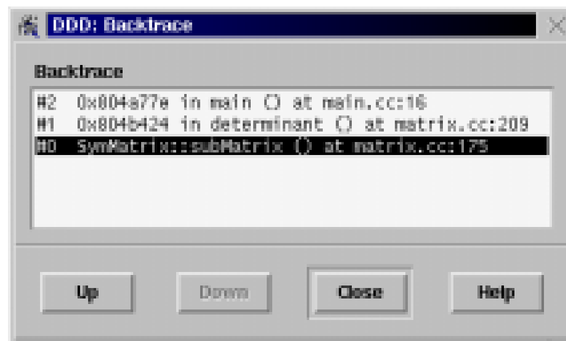


Abbildung 6.2 *Mit dem* BACKTRACE-*Dialog können Sie alle Aufrufebenen zurückverfolgen.*

Untersuchung von Variablen

Dabei wollen Sie sicher auch die Werte der verschiedenen Variablen überprüfen. Bei Standardtypen geht das am einfachsten, indem Sie mit dem Mauszeiger darauf deuten und einen Augenblick warten. Sofort erscheint ein kleines Hilfefenster, das den Wert der jeweiligen Variablen (oder auch Konstanten) enthält. Diese Information wird zudem gleichzeitig in der Statuszeile ausgegeben.

Wollen Sie alle lokalen Variablen verfolgen, genügt die Auswahl des Menüpunktes DATA | DISPLAY LOCAL VARIABLES. In einem Kästchen im Datenfenster werden Sie dann ständig über die aktuellen Werte informiert. Für Grafik-Freunde ist sogar eine Ausgabe von einer oder mehrerer Variablen als Plot über die gleichnamige Schaltfläche der Werkzeugleiste möglich (sofern Sie *gnuplot* installiert haben). Sie müssen lediglich die zu untersuchende Variable im Datenfenster oder im Quelltextfenster markiert haben.

Natürlich findet sich auch im *DDD* die dauerhafte Verfolgung von Variablenwerten wieder, die wir beim *gdb* als `display`-Kommando kennen gelernt haben. Eine Möglichkeit bietet das Kontextmenü. Klicken Sie über der interessierenden Variablen im Quelltextfenster die rechte Maustaste; hier können Sie nun sowohl eine Anzeige des Wertes selbst als auch, bei Zeigern, des dereferenzierten Inhalts aktivieren. Die Ausgabe erfolgt in einem Kästchen des Datenfensters.

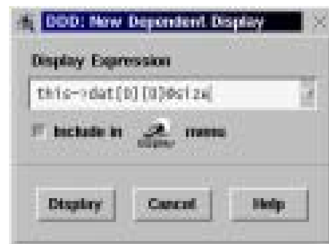


Abbildung 6.3
Für Felder legen Sie am besten ein eigenes Display-Format fest.

Die Anzeige von Feldern geht nicht ganz so automatisch. Hier ist wieder Ihr *gdb*-Know-how gefragt. Auf Seite 450 hatten wir festgestellt, dass man mehrere Einträge eines Feldes zusammen ausgeben kann, indem man das @-Zeichen und die Anzahl der Werte hinter die Angabe des ersten Wertes setzt. Diese Möglichkeit wählen wir auch hier: Klicken Sie auf die Schaltfläche DISPLAY der Werkzeugleiste und halten Sie sie für einen Moment gedrückt. Aus dem sich öffnenden Menü wählen wir OTHER. Dadurch gelangen wir zu einem Dialog (Abbildung 6.3), in dem wir unsere Eingabe, beispielsweise

```
this->dat[0][0]@size
```

vornehmen können. Anschließend wird uns das Feld im Datenfenster gezeigt (Abbildung 6.4). In großen Feldern, in denen derselbe Wert mehr als zehn Mal hintereinander vorkommt, wird dieser nur einmal angezeigt und seine Häufigkeit durch ein nachgestelltes »`<Nx>`« deutlich gemacht.

Bearbeiten und Übersetzen aus dem Debugger

Wenn Sie beim Debuggen Ihren Fehler entdeckt haben, können Sie natürlich zu Ihrer Entwicklungsumgebung zurückkehren, die Codestelle korrigieren und das Programm übersetzen. Gehören Sie jedoch zu

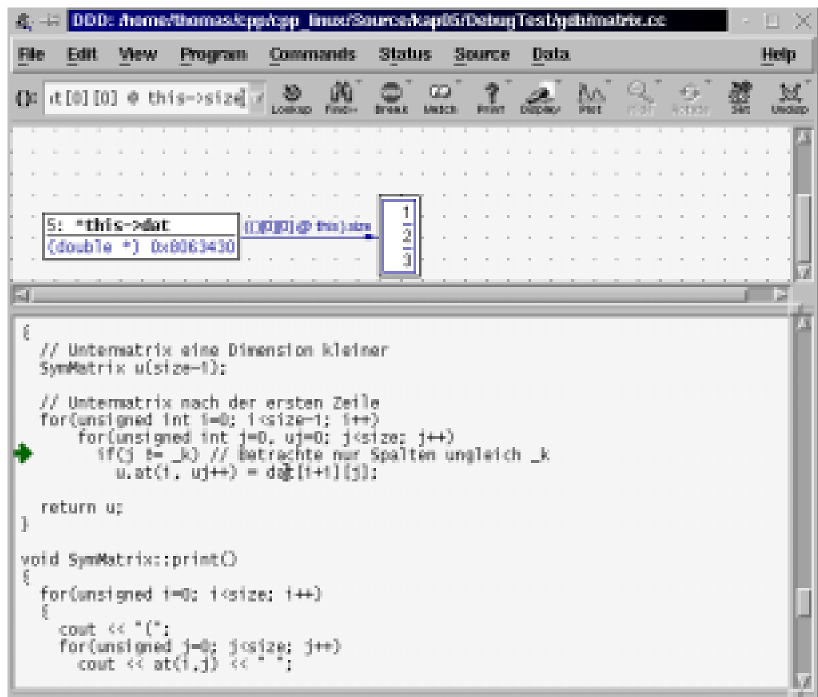


Abbildung 6.4 *Die gewünschten Feldelemente erscheinen im Datenfenster.*

den »Kommandozeilenprogrammierern«, die alle Arbeit an ihren Programmen von der Shell aus erledigen, können Sie sich diesen Umweg sparen. Über EDIT aus der Kommandoleiste können Sie ein Fenster öffnen, das Ihnen das Editieren der aktuellen Datei mittels des *vi*-Editors (siehe Seite 382) erlaubt. Änderungen erscheinen sofort nach Schließen des Editors im Quelltextfenster. Wenn Sie nun noch auf MAKE klicken, wird die geänderte Datei (bei einem korrekten Makefile) sofort übersetzt und das Programm aktualisiert. Der *DDD* merkt übrigens beim Starten eines Programms, ob es in einem anderen Prozess neu gebaut wurde und liest die Symboltabelle gegebenenfalls nochmals ein.

Hilfe

Und wenn Sie mal gar nicht mehr weiter wissen, gibt Ihnen HELP | WHAT NOW? immer einen freundlichen Hinweis, was Sie als Nächstes tun könnten. Um mehr über die Arbeit mit dem *DDD* zu erfahren, finden Sie unter HELP | DDD REFERENCE ein ausführliches Handbuch (das zudem der Installation auch im PostScript-Format beiliegt). Bei jedem Start begrüßt Sie der *DDD* außerdem mit einem »Tip of the day«; anhand dieser Tipps können Sie auch viel über die effiziente Bedienung dieses Werkzeugs lernen.

Die grafische Datenanzeige

Insgesamt erkennen Sie, dass der *DDD* alle Fähigkeiten des *gdb* in bequemerer Form zugänglich macht, gleichzeitig aber einige eigene Funktionalität mitbringt. Dieser wollen wir uns nun widmen.

Beispiel: Geburtstagserinnerung

Als Beispiel verwende ich diesmal nicht die Determinantenberechnung aus dem letzten Abschnitt; für diese Funktionen brauchen wir etwas komplexere Datenstrukturen. Wir sehen uns daher das Programm *birthcontrol* aus Abschnitt 6.1 (ab Seite 418) an, das Sie an Geburtstage Ihrer Familie und Freunde erinnern soll. Die zentrale Rolle dabei spielt die Klasse `BirthList`. Werfen wir einen Blick auf die Deklaration (die schon viel von der Definition enthält):

```
 1: // Datei: birthlist.h
 2: #include <string>
 3: #include "date.h"
 4:
 5: using std::string;
 6:
 7: //-----------------------------------------------
 8: // Klasse fuer Elemente der Liste
 9: //-----------------------------------------------
10: struct BirthListElement
11: {
12:   BirthListElement* next;
13:   string            name;
14:   string            surname;
15:   Date              date;
16:
17:   // Standardkonstruktor
18:   BirthListElement() :
19:     next(0) {}
20:
21:   // Spezieller Konstruktor
22:   BirthListElement(const string& _name,
23:     const string& _surname, unsigned short _day,
24:     unsigned short _month, unsigned short _year) :
25:     next(0),
26:     name(_name),
27:     surname(_surname),
28:     date(_day, _month, _year)
29:     {}
30: };
31:
32: //-----------------------------------------------
33: // Listenklasse
```

```
34: //-----------------------------------------------
35: class BirthList
36: {
37: private:
38:   BirthListElement* first;
39:   BirthListElement* last;
40:   int size;
41:
42: public:
43:   BirthList() :
44:     first(0), last(0), size(0) {}
45:
46:   virtual ~BirthList();
47:
48:   bool empty() const
49:     { return (size == 0); }
50:
51:   int getSize() const
52:     { return size; }
53:
54:   void pushBack(const string& _name,
55:     const string& _surname, unsigned short _day,
56:     unsigned short _month, unsigned short _year);
57:
58:   void popFront();
59:
60:   BirthListElement* front()
61:     { return first; }
62:
63:   int load(const string& filename);
64:
65:   bool check(const Date& d);
66: };
```

Die einzelnen Daten werden in Objekten vom Typ `BirthListElement` abgelegt (Zeile 10-30). Wie in Abbildung 3.12 (Seite 239) speichern wir auch hier die Daten in Form einer einfach verketteten Liste. Das nächste Listenelement wird durch den Zeiger `next` ausgedrückt. Um aber bequem auf die Liste zugreifen zu können, speichern wir in der Klasse `BirthList` (Zeile 35-66) sowohl einen Zeiger auf das erste (`first`) als auch auf das letzte Element (`last`).

Wie können wir die enthaltenen Daten sichtbar machen? Sobald Sie mit der linken Maustaste im Quelltextfenster auf einen Bezeichner klicken, erscheint dieser im Argumentfeld (das ist die Combobox unterhalb des Hauptmenüs, das mit einem Klammerpaar davor gekennzeich-

Abbildung 6.5
Die grafische Datenanzeige umfasst auch eingebettete Objekte sowie Zeiger auf weitere Objekte.

net ist). Alle `print`- oder `display`-Befehle, die Sie nun über das Kontextmenü oder die Schaltfläche der Werkzeugleiste aktivieren, nehmen darauf Bezug. (Daher taucht bei diesen auch immer das leere Klammernpaar auf.) Die so ausgeführten `display`-Kommandos erscheinen im Datenfenster in Form von Kästchen. Dort können Sie dann weitere Details in Erfahrung bringen.

Expandieren geschachtelter Datenstrukturen

Bei Datenstrukturen, die wieder andere Strukturen enthalten (wie unser `BirthListElement`), werden die Daten zunächst in knapper Form angezeigt, etwa als Pünktchen »...« oder als Wert des Zeigers. Um diese zu expandieren, wählen Sie zunächst die gewünschte Struktur aus (durch einen Klick, durch mehrere Klicks mit festgehaltener ⇑-Taste oder durch Umranden der Kästchen mit einem Fangrechteck mittels Ziehen mit der linken Maustaste). Anschließend öffnen Sie das Kontextmenü und klicken auf SHOW ALL. Damit weisen Sie die Anzeige an, alle Verschachtelungen aufzulösen und sämtliche Daten darzustellen. Ein mögliches Resultat können Sie in Abbildung 6.6 sehen.

Über den Schalter SHOW beziehungsweise HIDE der Werkzeugleiste lässt sich das Expandieren noch selektiver gestalten. So zeigt etwa SHOW MORE die Inhalte der gerade verborgenen Datenstruktur an,

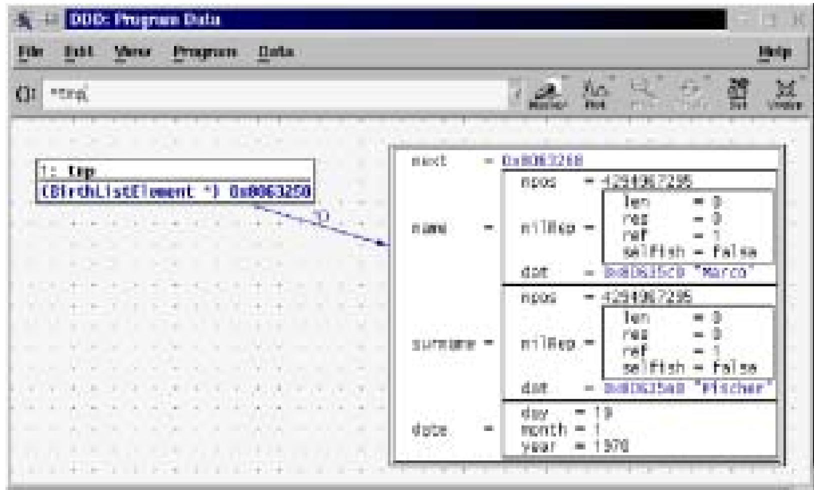


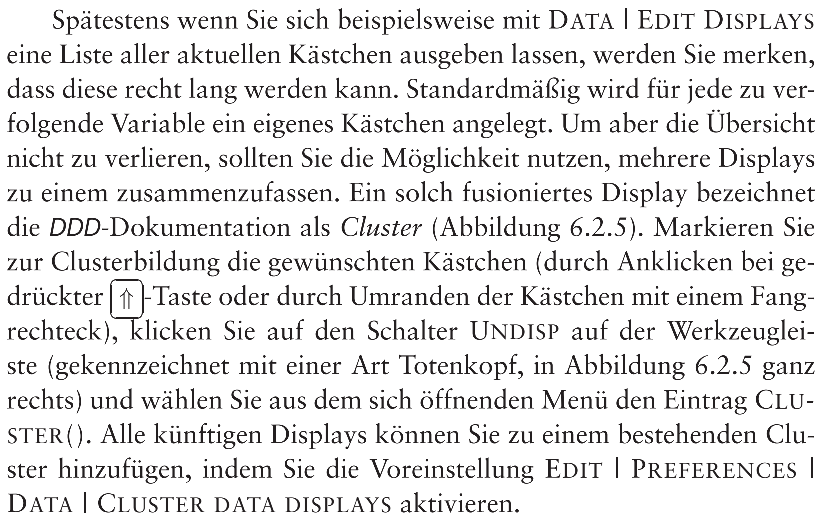
Abbildung 6.6
Mit SHOW ALL *können Sie sämtliche Daten einer verschachtelten Struktur expandieren.*

aber nicht die Inhalte der in ihr enthaltenen. Mit SHOW JUST werden alle Details eingeschachtelter Strukturen verborgen und nur die aktuelle Ebene angezeigt. Als Gegenstück dazu verbirgt HIDE alles.

Pfeile zwischen Displays

Die Pfeile zwischen den Kästchen (den so genannten *Displays*) verraten die Beziehung zwischen ihnen. In Abbildung 6.6 entspricht der rechte Kasten beispielsweise einer Dereferenzierung des Zeigers im linken, angedeutet durch `*()` über dem Pfeil. Wenn Sie jetzt auf den Eintrag `next` doppelt klicken, erscheint ein neuer Kasten mit dessen Inhalt; über dem Pfeil steht dann `next`.

Cluster von Displays



Spätestens wenn Sie sich beispielsweise mit DATA | EDIT DISPLAYS eine Liste aller aktuellen Kästchen ausgeben lassen, werden Sie merken, dass diese recht lang werden kann. Standardmäßig wird für jede zu verfolgende Variable ein eigenes Kästchen angelegt. Um aber die Übersicht nicht zu verlieren, sollten Sie die Möglichkeit nutzen, mehrere Displays zu einem zusammenzufassen. Ein solch fusioniertes Display bezeichnet die *DDD*-Dokumentation als *Cluster* (Abbildung 6.2.5). Markieren Sie zur Clusterbildung die gewünschten Kästchen (durch Anklicken bei gedrückter ⇑-Taste oder durch Umranden der Kästchen mit einem Fangrechteck), klicken Sie auf den Schalter UNDISP auf der Werkzeugleiste (gekennzeichnet mit einer Art Totenkopf, in Abbildung 6.2.5 ganz rechts) und wählen Sie aus dem sich öffnenden Menü den Eintrag CLUSTER(). Alle künftigen Displays können Sie zu einem bestehenden Cluster hinzufügen, indem Sie die Voreinstellung EDIT | PREFERENCES | DATA | CLUSTER DATA DISPLAYS aktivieren.

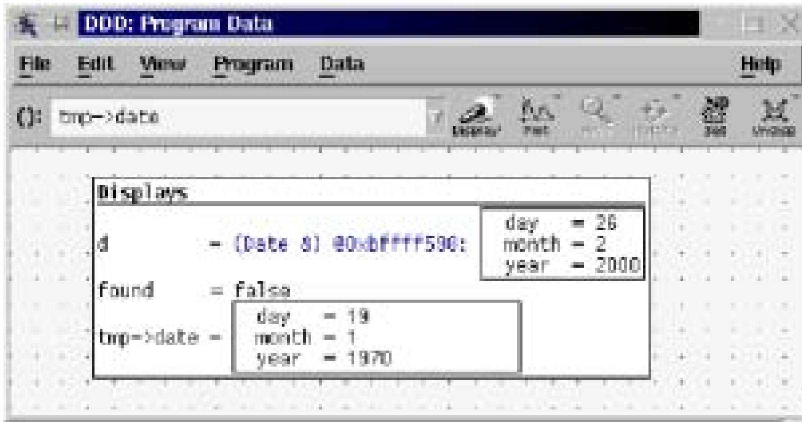


Abbildung 6.7
Mit einem Cluster lassen sich mehrere Kästchen in einem zusammenfassen.

Fazit

Wie Sie gesehen haben, ist der *DDD* mehr als nur ein Frontend zum *gdb*. Er verfügt über eine Reihe eigener Fähigkeiten, von denen ich Ihnen hier leider nur einen kleinen Teil zeigen konnte. Lassen Sie sich also bei Ihrer Arbeit mit dem *DDD* durch den »Tip of the day« oder einen Blick ins Handbuch dazu anregen, weitere Möglichkeiten zu erkunden. Sie werden sehen: Es lohnt sich!

6.2.6 Fehlervermeidung durch die Überprüfung von Vorbedingungen

Bei den Fehlerquellen, die wir in Abschnitt 6.2.1 untersucht haben, blieb ein Typ völlig unberücksichtigt: falsch benutzte Methoden (oder Funktionen). Denn jede Methode, die nicht gerade konstant ist, ändert den Zustand des Objekts oder des ganzen Programms. Daher muss die Übergabe ungültiger Parameter oder die falsche Reihenfolge des Aufrufs zu einem undefinierten Zustand führen, der sich früher oder später auf das ganze Programm fatal auswirkt.

Vor- und Nachbedingungen

Um die Aufrufspezifikation exakt zu formulieren, sollte man Vor- und Nachbedingungen für jede Methode festlegen. Vorbedingungen sind die Erwartungen, die die Methode in die Parameter und den Aufrufkontext setzt (dass also zum Beispiel ein Parameter größer als null ist). Nachbedingungen drücken aus, was der Aufrufer von der Methode erwarten darf.

Verträge

Auf diese Weise wird die Beziehung zwischen einer Funktion und deren Benutzer auf eine klar definierte Grundlage gestellt. Eine solche kann man auch als *Vertrag* zwischen beiden auffassen. Als einer der ersten hat Bertrand Meyer in seinem sehr lesenswerten Buch