Datenstrukturen visualisieren und animieren mit DDD

Andreas Zeller

Universität Passau
Lehrstuhl für Software-Systeme
Innstraße 33, 94032 Passau
zeller@acm.org

Zusammenfassung Der graphische Debugger *DDD* ist mit mehr als 250.000 Anwendern eins der verbreitetsten Werkzeuge zur Softwarevisualisierung. Neben den üblichen Debugger-Funktionalitäten ermöglicht DDD die *Visualisierung von Datenstrukturen* im laufenden Programm. DDD kann verzeigerte Strukturen (wie Listen oder Bäume) als Graphen darstellen, aber auch numerische Felder in zwei- oder dreidimensionalen Plots darstellen und im Programmlauf animieren. Dieser Beitrag beschreibt die technischen Grundlagen der Visualisierung, die verwendeten Layoutverfahren, sowie die Animation von Algorithmen.

Schlüsselwörter: Visuelles Debuggen, Visualisierung der Daten und Abläufe in Anwendungen, Visualisierung von Datenstrukturen, Algorithmenanimation.

1 Einführung: Visuelles Debuggen

Trotz aller Fortschritte in den frühen Phasen der Softwareentwicklung gehört die *Fehlersuche* weiterhin zum täglichen Brot der Programmierer. *Interaktive Debugger* haben sich als Universal-Werkzeuge zur Fehlersuche etabliert. Mit einem Debugger kann man

- das Programm in einer definierten Umgebung *ausführen*,
- das Programm unter bestimmten Bedingungen *anhalten lassen*,
- den Zustand des angehaltenen Programms *untersuchen* und
- den Zustand des angehaltenen Programms *verändern*.

Wenn es um die Untersuchung des Programmzustands geht, ist der zentrale Aspekt die *Datenausgabe*. Traditionelle Debugger geben ihre Daten als *Text* aus – so etwa der GNU-Debugger GDB:

```
(gdb) print *tree
*tree = {value = 7, _name = 0x8049e88 "Ada", _left = 0x804d7d8,
  _right = 0x0, left_thread = false, right_thread = false,
  date = {day_of_week = Thu, day = 1, month = 1, year = 1970,
   _vptr. = 0x8049f78 <Date virtual table>}, static shared = 4711}
(gdb) _
```

Abb. 1. Textuelle Datenausgabe mit GDB

Auch wenn moderne Debugger die Gesamtausgabe nicht mehr als Fließtext gestalten, werden doch die Daten nach wie vor als *Text* ausgegeben. Für einzelne Daten ist diese textuelle Darstellung gewiß angemessen – wie anders wollte man eine Zeichenkette

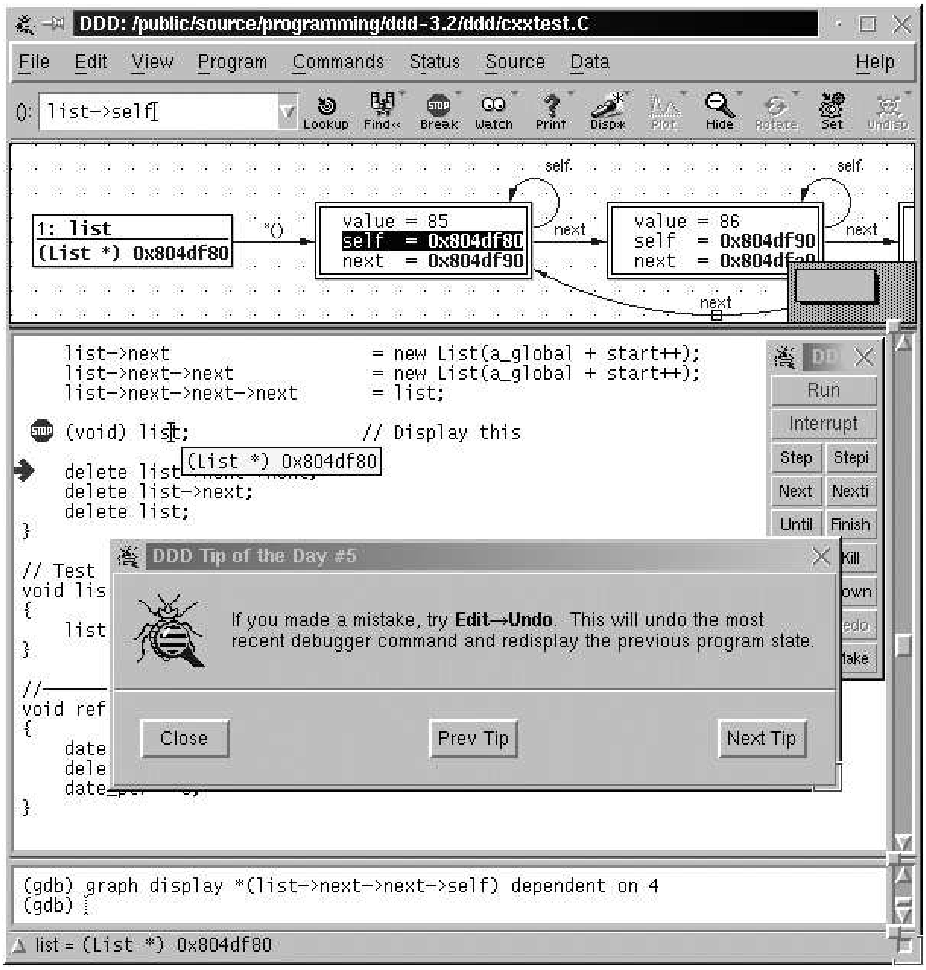


Abb. 2. Darstellung von Datenstrukturen in DDD

schon betrachten? Schwierig wird es aber, wenn es um *Beziehungen* zwischen Daten geht – Referenzen, Zeiger, Indizes und dergleichen. Worauf etwa zeigt `tree→left`? Natürlich kann der Anwender dies individuell abfragen. Wenn aber zwei Zeiger denselben Wert haben, kann der Anwender dies nur durch mühsames Vergleichen der Adressen bestimmen.

Getreu dem Motto, daß „der Mensch bildliche Eindrücke besonders effizient verarbeiten kann und konkrete Objekte besser erfassen kann als abstrakte Beschreibungen“ [2], haben wir einen Debugger konstruiert, der über *eingebaute Datenvisualisierung* verfügt – den *Data Display Debugger* oder kurz DDD.

DDD ist technisch realisiert als *Front-End* zu herkömmlichen Kommandozeilen-Debuggern. DDD unterstützt derzeit GDB, DBX, LADEBUG, XDB, JDB, den Python-Debugger und den Perl-Debugger; die Oberfläche (Abbildung 2) bietet allen gewohnten Komfort einschließlich einer detaillierten Zustands- und Kontext-sensitiven Hilfe. DDD ist als offizielle GNU-Software im Quelltext frei erhältlich und wird heute von über 250.000 Anwendern zur Programmentwicklung eingesetzt; die Liste der Anwender reicht von Adobe und Boeing bis zu Texas Instruments und Xerox.

Im folgenden zeigen wir, wie sich DDD als Universalwerkzeug zur dynamischen Softwarevisualisierung einsetzen läßt. Abschnitt 2 beschreibt zunächst die Visualisierung von Datenstrukturen aus dem laufenden Programm; die Layoutregeln werden in Abschnitt 3 gesondert erläutert. Abschnitt 4 diskutiert die alternative Darstellung numerischer Werte als Plots. Als Debugger kann DDD dargestellte Daten während des Programmlaufs auf den neuesten Stand bringen; Abschnitt 5 zeigt, wie sich dies zur Algorithmenanimation einsetzen läßt. Verwandte Arbeiten sind in Abschnitt 6 behandelt; Abschnitt 7 schließt mit Zusammenfassung und Ausblick.

2 Visualisierung von Datenstrukturen

Grundidee der Visualisierung in DDD ist, jedem individuellen Datum einen eigenen Bereich, eine sogenannte *Box*, zuzuweisen. Ineinander geschachtelte Strukturen werden durch ineinander geschachtelte Boxen dargestellt, die zum Ausblenden durch Platzhalter ersetzt werden können. Ein solches komplettes Einzel-Datum wird *Display* genannt.

Die Stärke von DDD liegt im Visualisieren der *Beziehungen* zwischen Displays. Hierfür werden die einzelnen Displays durch *gerichtete Kanten* zu einem Graph zusammengefaßt. Wird ein neues Datum d' aus einem bestehenden Datum d angezeigt – etwa durch *Dereferenzieren* eines Zeigers in d –, so stellt DDD die Beziehung durch eine Kante von d nach d' dar. Auf diese Weise lassen sich komplexe Datenstrukturen darstellen (Abbildung 3).

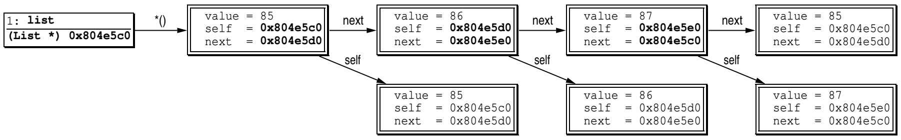


Abb. 3. Darstellung verketteter Listen ohne Aliaserkennung

Normalerweise zeigt auf jedes neu erschaffene Datum maximal eine Kante – die Darstellung ist also auf Baumstrukturen beschränkt. Durch Aktivieren einer besonderen *Aliaserkennung* kann DDD jedoch auch komplexere Datenstrukturen darstellen. Grundidee ist, die Displays, die an derselben Adresse im Speicher stehen (*Aliase*), zu *verschmelzen*, um so mehrere Verweise auf ein Datum zu ermöglichen.

Als Beispiel dienen erneut die Daten von Abbildung 3. Das Vergleichen der Zeiger-Werte macht deutlich, daß die `self`-Zeiger jeweils auf die Displays selbst zeigen; außerdem verweist der letzte `next`-Zeiger wieder auf das erste Element der Liste. Nach Einschalten der Aliaserkennung erfragt DDD für alle angezeigten Daten deren Speicheradressen und verschmilzt die Boxen mit gleicher Adresse. Das Ergebnis ist in Abbildung 4 auf der nächsten Seite zu sehen – das Bild zeigt deutlich, welche Zeiger auf welche Daten verweisen, ohne daß der Anwender erst deren Adressen textuell verglei-chen müßte.

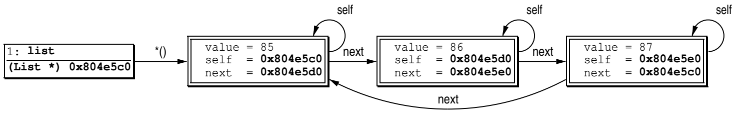


Abb. 4. Darstellung verketteter Listen mit Aliaserkennung

Die Aliaserkennung erlaubt das Darstellen auch komplexerer Datenstrukturen wie im Lehrbuch. Abbildung 5 zeigt, wie sich ein gefädelter Baum in DDD präsentiert.

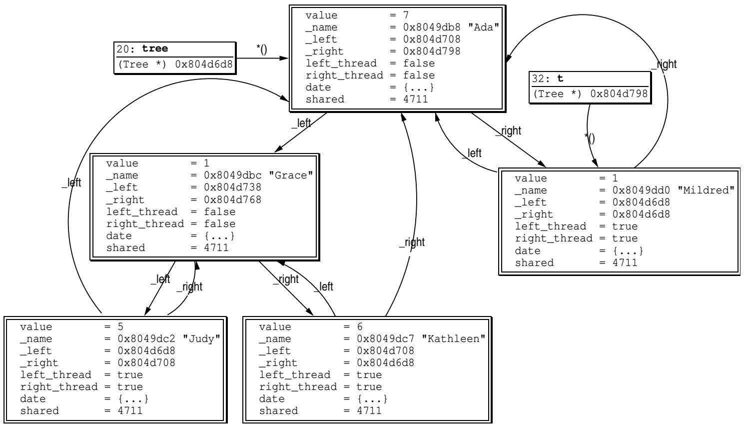


Abb. 5. Ein gefädelter Baum in DDD

Neben dem einfachen Dereferenzieren von Zeigern unterstützt DDD beliebige andere Verweis-Operationen. Erzeugt der Benutzer etwa einen Tabelleneintrag $t[i]$ aus einem Index i, so wird eine Kante von i nach $t[i]$ gezogen. Die Operationen werden in einem Menü gespeichert und dienen so zur Erkundung und Darstellung beliebiger Verweise.

3 Layoutalgorithmen

In DDD erkundet der Programmierer Datenstrukturen *interaktiv* und *inkrementell* – er entscheidet selbst, welchen Querverweisen er folgen möchte. Damit entfällt die Notwendigkeit, eine Datenstruktur am Stück zu erkennen und zu layouten. DDD kennt lediglich vier einfache Regeln, anhand der neue Displays positioniert werden, dargestellt in Abbildung 6 auf der nächsten Seite.

Ausgangslage ist, daß der Anwender ein neues Display d' aus einem bestehenden Display d erzeugt – etwa, indem er d dereferenziert und d' erhält. Dann lassen sich vier Fälle unterscheiden:

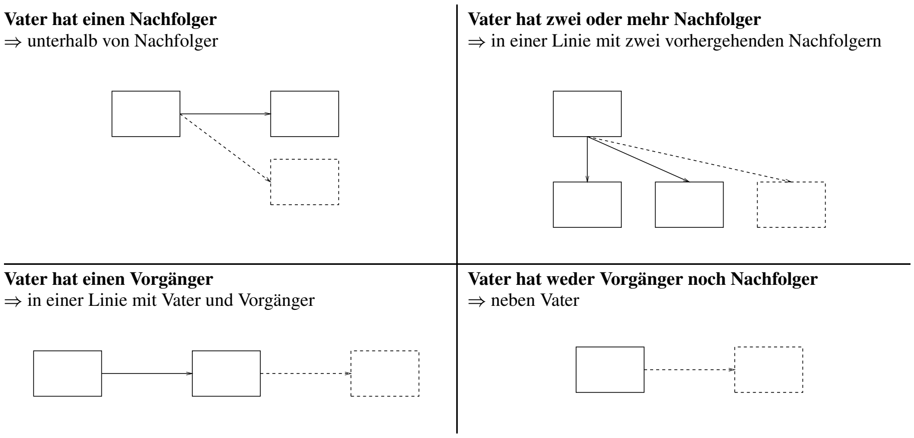


Abb. 6. Positionierungsregeln in DDD. Das neu zu positionierende Display ist gestrichelt.

- Hat d (der „Vater") bereits einen Nachfolger, so wird d' neben d plaziert. Dies ermöglicht das „Auffächern" mehrerer Nachfolger.
- Hat d mehr als einen Nachfolger, wird d' in Verlängerung des Abstands zwischen den beiden letzten Nachfolgern plaziert. Dies führt dazu, daß alle Nachfolger aneinander ausgerichtet sind.
- Hat d keinen Nachfolger, aber einen Vorgänger, wird d' in Verlängerung des Abstands zwischen d und dessen Vorgänger plaziert. Hiermit werden verkettete Strukturen ausgerichtet.
- Hat d weder Vorgänger noch Nachfolger, wird d' neben d plaziert. Die Vorgabe für diese Ausrichtung ist „rechts von d"; der Benutzer kann dies durch Rotieren des Graphen ändern.

Nach der ursprünglichen Positions-Vorgabe kann der Anwender das Layout nach Belieben manuell verändern. DDD bietet auch die Möglichkeit, den kompletten Graphen am Stück neu zu layouten. Hierfür setzt DDD das Verfahren von Sugiyama und Misue [8] ein. In jedem Layoutvorgang geht die bisherige Ortsinformation verloren – jedes Display erhält einen neuen Platz. Daher muß sich der Anwender komplett neu orientieren, was belastend wirkt. Unserer Erfahrung nach bevorzugen die Benutzer das manuell-inkrementelle Layout.

Nichtdestotrotz bleibt auch das inkrementelle Layout mühsam, wenn große Strukturen Stück für Stück erkundet werden müssen. Für künftige DDD-Versionen ist deshalb eine *automatische Entfaltung* von Datenstrukturen geplant: Nach dem Positionieren der ersten zwei oder mehr Displays werden die Operationen solange auf die Nachfolger angewandt, bis ein Fixpunkt erreicht ist. Auf diese Weise lassen sich beliebige Datenstrukturen durch einfaches automatisches Wiederholen der Zugriffsmechanismen entfalten.

Die konkrete Darstellung der Daten und Beziehungen in DDD hat den Nachteil, daß wenig Raum für abstrakte (Text-)Daten bleibt. Neuere DDD-Versionen bieten deshalb die Möglichkeit, Displays in *Cluster* zusammenzufassen. Werden grundsätzlich alle Displays geclustert, erhält man eine herkömmliche platzsparende Textdarstellung wie auch in traditionellen Debuggern.

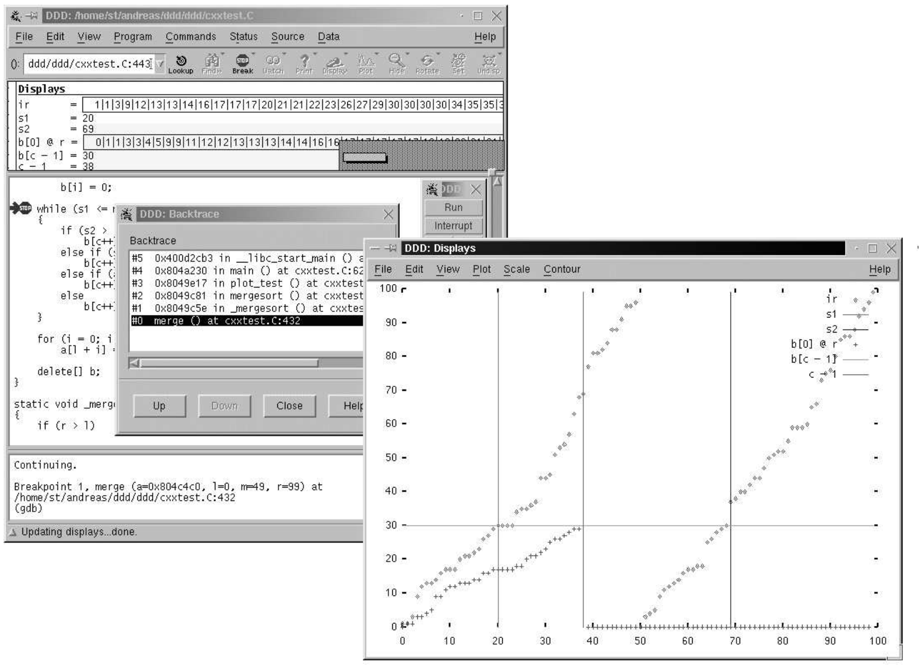


Abb. 7. Plots in DDD

4 Visualisierung numerischer Werte

Neben der Darstellung von Datenstrukturen ermöglicht es DDD, numerische Werte in Form von *Plots* darzustellen. Hierfür greift DDD auf die Dienste des externen Werkzeugs GNUPLOT zurück. Das Grundverfahren ist ganz einfach:

1. Der Benutzer wählt ein Datenfeld aus, das geplottet werden soll.
2. DDD erfragt vom Kommandozeilen-Debugger die Werte und sendet sie an GNUPLOT.
3. DDD führt die von GNUPLOT generierten Zeichenanweisungen in einem eigenen Fenster aus.

Abbildung 7 zeigt einen so generierten Plot. Insgesamt kann DDD folgende Datentypen als Plot darstellen:

– Eindimensionale Felder werden als 2D-Plot über den Index gezeichnet.
– Zweidimensionale Felder werden als 3D-Plot über beide Indizes gezeichnet.
– Einfache numerische Werte werden als Linien gezeichnet. Wird der Wert zusammen mit einem Feld gezeichnet und liegt der Wert im Indexbereich des Feldes, so zeichnet DDD den Wert als vertikale x-Konstante, ansonsten als horizontale y-Konstante. Der Benutzer kann die Ausrichtung selbst ändern.
– Zusammengesetzte Werte (Cluster, Records, usw.) werden in einem Plot zusammengefaßt. DDD berücksichtigt hierbei alle auftretenden numerischen Werte – Zeichenketten oder Zeiger werden übergangen.

Der letzte Punkt ist besonders wichtig, wenn man mehrere Werte zueinander in Beziehung setzen will. In Abbildung 7 etwa sehen wir einen Ausschnitt aus der *merge*-Funktion des *mergesort*-Algorithmus. Die äußeren vertikalen Linien sind die Indizes über die beiden Hälften, aus denen gemischt wird; die mittleren Linien zeigen die obere Grenze des bereits gemischten Feldes. DDD stellt Menüs zur Verfügung, mit denen die wichtigsten Aspekte des Plots verändert werden können; die Plots können auch gedruckt und an andere Programme exportiert werden.

Als besondere Eigenschaft erlaubt es DDD, die *Historie* eines Wertes zu plotten – also die Kette der früheren Werte während der Programmausführung. Bei jedem Programmhalt (etwa beim schrittweisen Ausführen) merkt sich DDD den dargestellten Programmzustand einschließlich aller Variablenwerte in einem *Historien-Speicher;* der Anwender kann durch *Undo* und *Redo* nach Belieben frühere Programmzustände erneut darstellen. Dies ist besonders praktisch, wenn man frühere Werte von Variablen noch einmal sehen möchte. Beim Plotten der Historie wird nun die Folge der Variablenwerte über die Anzahl der Programmhalte geplottet. Hiermit lassen sich ungewöhnliche Wertentwicklungen „auf einen Blick“ erkennen.

5 Algorithmenanimation

Bei jedem Halt des Programms bringt DDD alle angezeigten Werte auf den neuesten Stand. Dies gilt für die dargestellten Datenstrukturen und Plots; auch die Alias-Analyse wird bei jedem Programmhalt neu gestartet.

Neben dem eigentlichen Ziel des Debuggens läßt sich diese Eigenschaft besonders gut nutzen, um *Algorithmen zu animieren* – ganz ohne zusätzliches Werkzeug, Hilfsfunktionen oder auch nur eine Neuübersetzung. Abbildung 8 etwa zeigt das Löschen von Elementen aus einer verketteten Liste. Nach jedem Löschen stoppt ein Haltepunkt das Programm, und DDD stellt den neuen Wert der Liste dar.

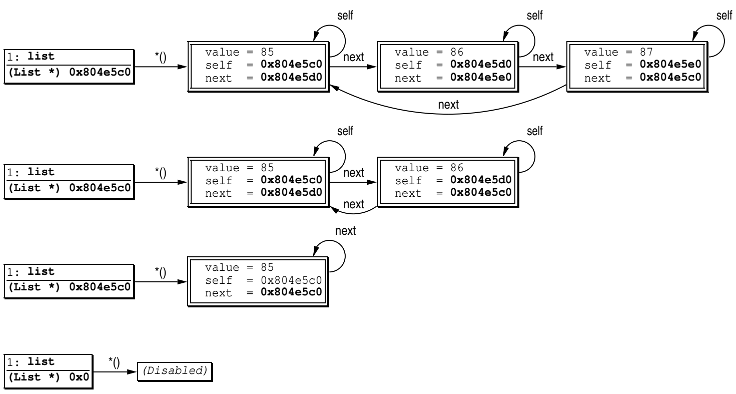
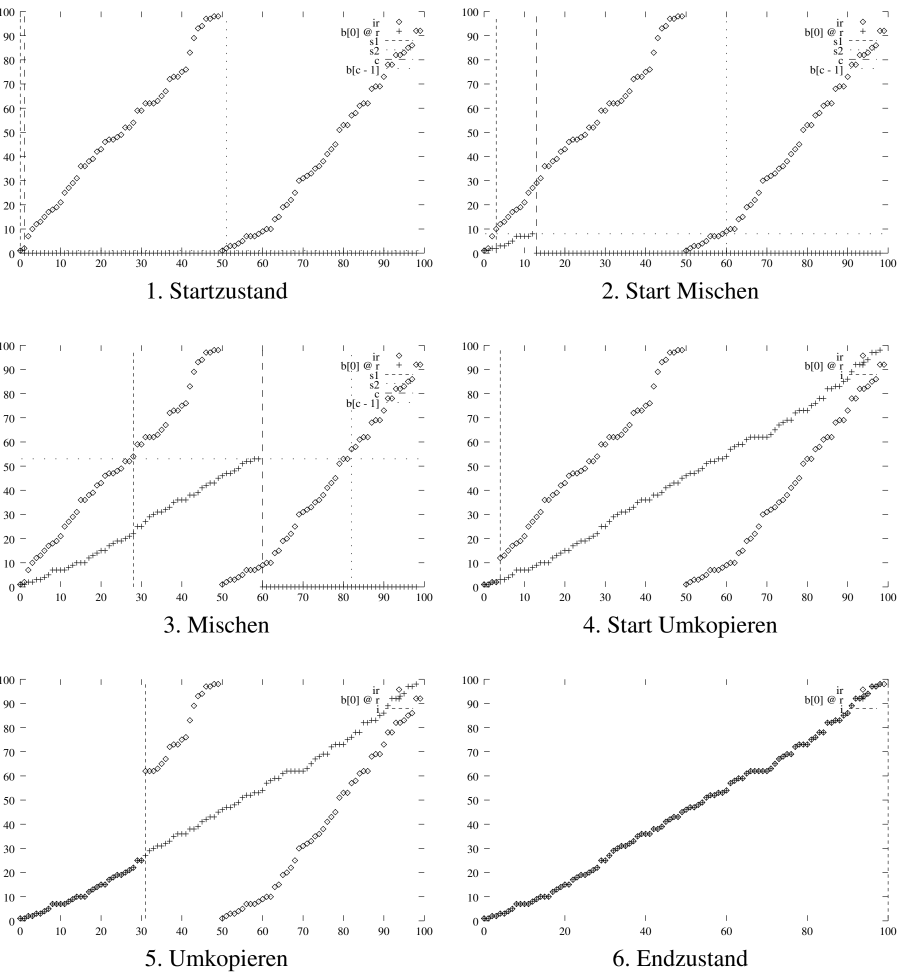


Abb. 8. Elemente aus einer verketteten Liste löschen[1]

[1] Für eine optimale Darstellung in diesem Beitrag zeigen wir statt DDD-Screenshots von DDD exportierte Grafiken. Alle Bilder können ebenso interaktiv am Bildschirm betrachtet werden.



1. Startzustand
2. Start Mischen
3. Mischen
4. Start Umkopieren
5. Umkopieren
6. Endzustand

Abb. 9. Animation eines *mergesort*-Algorithmus in DDD

Will man in der Lehre einen Algorithmus zeigen, ist insbesondere die *Undo*/*Redo*-Funktion nützlich. Durch *Undo*/*Redo* kann zwischen den einzelnen Zuständen in Abbildung 8 hin- und hergesprungen werden; gleichzeitig wird die entsprechende Anweisung im Programm dargestellt. Auch die Möglichkeit, eine *DDD-Sitzung* abzuspeichern – den Programmzustand einschließlich aller Displays und Haltepunkte – ist hier praktisch.

Arbeitet ein Algorithmus auf numerischen Werten, empfiehlt es sich, seine Daten als Plot zu animieren. Abbildung 9 etwa zeigt Ausschnitte aus dem Ablauf eines *mergesort*-Algorithmus, der bereits im DDD-Bild in Abbildung 7 gezeigt wurde. Deutlich ist zu erkennen, wie die Werte aus den beiden Hälften zusammengemischt und schließlich umkopiert werden.

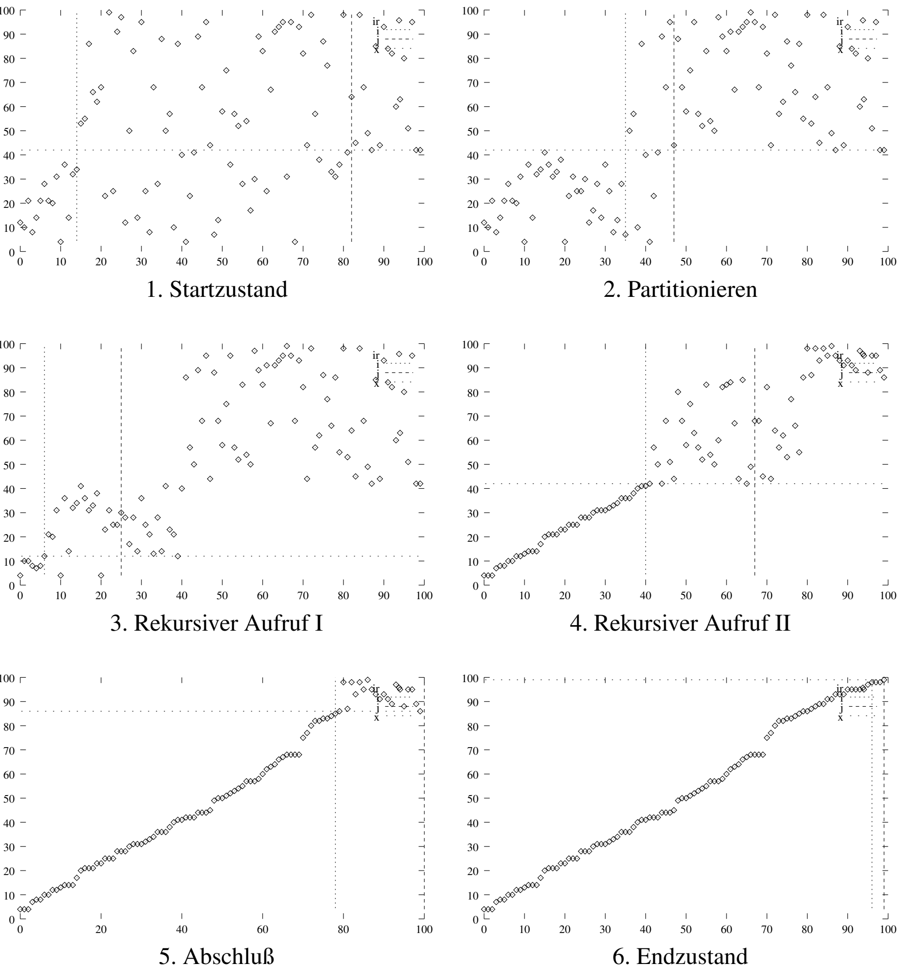


Abb. 10. Animation eines *quicksort*-Algorithmus in DDD

Ein zweites Beispiel zeigt Abbildung 10: Hier animiert DDD einen *quicksort*-Algorithmus. Dargestellt ist der Partitionierungsvorgang – alle Werte, die kleiner sind als das Pivot-Element (horizontale Linie), werden im Feld nach links bewegt; alle Werte, die größer sind, werden nach rechts bewegt. Die vertikalen Linien wandern von außen nach innen und stellen den Fortschritt des Verfahrens dar; treffen sich die Linien, ist die Partitionierung auf dieser Ebene abgeschlossen. In den späteren Schritten sieht man, wie der rekursive Aufruf der Partitionierung das gesamte Feld sortiert.

Noch wirkungsvoller werden diese Darstellungen, wenn man sie *vollautomatisch* ablaufen läßt. Hierzu setzt man einen Haltepunkt, der mit einem *continue*-Kommando gekoppelt ist. Beim Erreichen des Haltepunkts bringt DDD die Darstellung auf den neuesten Stand und führt das *continue*-Kommando aus, wodurch der Programmlauf fortgesetzt wird. Mit *bedingten Haltepunkten* kann die Ausführung an ausgewählten

Stellen unter definierten Bedingungen angehalten werden – etwa im erstmaligen rekursiven Aufruf oder bei Unterschreiten einer gewissen Schranke für den Index.

6 Verwandte Arbeiten

Neben DDD gibt es eine Reihe von Projekten, die ebenfalls Fehlersuche mit Visualisierung verbinden.

VCC [1] visualisiert Programme, indem es beim Übersetzen automatisch Aufrufe für eine Animations-Bibliothek in den Programmcode hinein übersetzt. Im Gegensatz zu DDD ist somit ein eigener Übersetzungsschritt erforderlich; auch ist VCC auf die Programmiersprache C beschränkt. VCC bietet Sichten für den Code, die aktive Funktion, den Baum der Funktionsaufrufe sowie Datenstrukturen.

ZStep95 [6] ist eine Visualisierungsumgebung für Common Lisp. Wie auch VCC fügt ZStep95 dem Programm spezielle Animations-Aufrufe hinzu; wie DDD speichert ZStep95 die Historie von Daten während eines Programmlaufs, die dann mittels eines „Videorekorders“ vor- und zurückgespult werden können. Im Gegensatz zu DDD, der nur dargestellte Daten speichert, merkt sich ZStep95 jegliche Veränderungen aller Daten, was erheblich Speicherplatz kostet.

Lens [7] ist wie DDD ein graphisches Debugger-Frontend, in diesem Fall für den DBX-Debugger. Lens bietet ein Entwicklungswerkzeug, in dem interaktiv Animationen zum Code erstellt werden können. Im Gegensatz zu DDD, wo die Animation über den Programmlauf erfolgt, müssen in Lens Animationsanweisungen manuell eingefügt werden. Dafür erlaubt Lens vielfältige Eingriffe in die Animation – etwa Farben, Strukturen, usw. – die bei DDD fest vorgegeben sind.

DEET [5] ist ebenfalls eine graphische Erweiterung für Kommandozeilen-Debugger. In DEET liegt der Schwerpunkt auf *Einfachheit:* Für die graphische Datendarstellung werden externe Layouter und Programme benutzt. Da DEET keine Alias-Analyse durchführt, ist die Darstellung auf Baumstrukturen beschränkt (die aber im Gegensatz zu DDD automatisch entfaltet werden); ein interaktives Layout ist in DEET nicht möglich.

DUEL [4] ist eine spezielle Programmiersprache für Debugger, mit der Daten auf einer höheren Abstraktionsstufe analysiert werden können. So bietet DUEL Quantoren, Filter und Mengenoperationen. Die Erweiterung XDUEL visualisiert die aus DUEL gewonnenen Daten. Der Hauptvorteil von DUEL ist gleichzeitig sein Nachteil: Die speziellen DUEL-Operatoren und -Befehle lösen sich von der Programmiersprache und erschweren so Anfängern die Bedienung.

ISVL [3] ist eine Arbeitsumgebung zur verteilten kollaborativen Fehlersuche. Die Visualisierungskomponente visualisiert eine Ausführungshistorie (ähnlich ZStep95), die dann über einen Web-Server einer Gruppe von Programmierern verfügbar gemacht wird. ISVL ist leider auf die Programmiersprache PROLOG beschränkt.

Von den beschriebenen Projekten ist keines über einen frühen Prototyp-Status hinausgekommen. Es wäre jedoch wünschenswert, wenn zukünftige praktisch eingesetzte Werkzeuge die hier realisierten Anregungen aufgriffen; auch DDD böte sich als Plattform an.

7 Zusammenfassung und Ausblick

DDD ist ein Universalwerkzeug zur dynamischen Softwarevisualisierung. DDD kann verzeigerte Strukturen (wie Listen oder Bäume) als Graphen darstellen, aber auch numerische Felder in zwei- oder dreidimensionalen Plots darstellen und im Programmlauf animieren. Herausragend in DDD ist der schnelle Zugang zu visualisierten Daten – es sind weder aufwendige Neuübersetzungen noch komplexe Befehlsstrukturen noch aufwendige Interaktionen vonnöten.

So angenehm und hilfreich die Datenvisualisierung in DDD sein mag, so schnell werden aber auch die *Grenzen* sichtbar. Ein Graph mit wenigen Dutzend Displays paßt noch gut auf den Bildschirm und ist leicht zu überblicken. In größeren Strukturen wird es aber schnell aussichtslos, Fehler durch simples Betrachten zu finden. Die Visualisierung von Daten zeigt hier dasselbe Problem wie jede andere Visualisierung: die *kognitive Aufnahmefähigkeit* des Menschen ist eine obere Schranke, die auch durch die beste Visualisierung nicht überwunden werden kann.

Man könnte versuchen, das Problem durch „intelligentere“ Darstellungen zu lösen – etwa durch Zusammenfassung von Daten oder Datentyp-spezifische Zeichenverfahren. Leider setzen solche Darstellungen Wissen über die Datenstrukturen voraus, das nicht ohne weiteres aus dem Programm abgeleitet werden kann. Wüßte DDD etwa, daß eine Datenstruktur einem gefädelten Baum entspricht, könnte DDD den Baum entsprechend zeichnen. Steht aber diese Intelligenz bereits zur Verfügung, könnte man genausogut entsprechende Prüfprogramme generieren, die jegliche Abweichung vom Lehrbuch-Baum aufdecken. Warum dann nicht gleich eine fertige, gut getestete Komponente wählen und Fehlern von vorneherein aus dem Weg gehen?

Zusammenfassend stößt man mit Visualisierung also offensichtlich an Grenzen, die nicht leicht zu überwinden sind; zusätzliche bereichsspezifische Intelligenz steht im Gegensatz zur angestrebten Universalität von Werkzeugen. Gerade bei Debuggern als Universalwerkzeugen, die schnell und unkompliziert die freie Erkundung des Programmzustands ermöglichen sollen, wird dieser Gegensatz deutlich.

Gelingt es aber, die Problemgröße klein genug zu halten, bringt Visualisierung großen Gewinn – in der Fehlersuche wie auch in der Lehre. Daß Debuggen Spaß machen kann, haben viele Menschen erst mit DDD gelernt, und das liegt nicht zuletzt an seinen universellen und eingängigen Visualisierungstechniken.

Danksagung. Holger Cleve, Torsten Robschink und die anonymen Gutachter gaben wertvolle Kommentare zu diesem Beitrag. Dorothea Krabiell (geb. Lütkehaus) ist Co-Autorin der ersten DDD-Fassung.

DDD, Dokumentation und Online-Verweise auf verwandte Projekte finden Sie unter

`http://www.gnu.org/software/ddd/` .

Literatur

1. BAEZA-YATES, R., G. QUEZADA und G. VELMADRE: *Visual Debugging and Automatic Animation of C Programs*. In: EADES, P. und K. ZHANG (Hrsg.): *Software Visualisation*. World Scientific Press, 1996.

2. DIEHL, S.: *GI-Workshop Softwarevisualisierung – Call for Papers*, Jan. 2000.
3. DOMINGUE, J. und P. MULHOLLAND: *Fostering Debugging Communities on the Web*. Communications of the ACM, 40(4):65–71, Apr. 1997.
4. GOLAN, M. und D. R. HANSON: *DUEL—a very high-level debugging language*. In: *Proceedings of the Winter USENIX Technical Conference*, S. 107–117, San Diego, California, USA, Jan. 1993.
5. HANSON, D. R. und J. L. KORN: *A Simple and Extensible Graphical Debugger*. In: *Proceedings of the USENIX 1997 Annual Technical Conference*, Bd. 183-174, Anaheim, California, USA, Jan. 1997.
6. LIEBERMANN, H. und C. FRY: *ZStep95: A Reversible, Animated Source Code Stepper*. In: STASKO, J., J. DOMINGUE, B. PRICE und M. BROWN (Hrsg.): *Software Visualization: Programming as a Multimedia Experience*. MIT Press, 1997.
7. STASKO, J. T. und S. MUKHERJEA: *Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger*. ACM Transactions on Computer-Human Interaction, 1(3):215–244, Sep. 1994.
8. SUGIYAMA, K. und K. MISUE: *Visualization of Structural Information: Automatic Drawing of Compound Digraphs*. IEEE Transactions on Systems, Man, and Cybernetics, SMC-21(4):876–892, Juli 1991.