Seminar Visualisierung in Informatik und Naturwissenschaften SS99 Visuelles Debugging

Dalibor Topić
topic@studcs.uni-sb.de

27.5.99

Zusammenfassung

Ein Überblick über visuelles Debugging im Laufe der Geschichte, am Beispiel von konkreten Systemen VCC, ZStep95, Lens und DDD.

Visuelles Debugging ist ein Thema, das von der Visualisierungs-Community etwas stiefmütterlich behandelt wird. Die Menge der Papers zum Thema ist relativ überschaubar, und die Menge der Tools, die den Sprung aus der Forschung in die Praxis schaffen, ebenfalls. Andererseits, schlägt man ein Entwicklermagazin wie Java Report oder C \C++ Users Journal auf, so scheinen die meisten Anzeigenkunden der Meinung zu sein, daß für visuelle Debuggingtools ein großer Markt existiert. Diese Firmen bieten eine breite Palette an visuellen Debuggern, Profilern, und Speicherfehler-Tools.

Ich beginne mit verschiedenen Definitionen für das Debugging.

1 Definitionen für Debugging

1.1 Definition von Steve McConnell [1]

Debugging is the process of identifying the root cause of an error and correcting it. It contrasts with testing, which is the process of detecting the error initially. On some projects, debugging occupies as much as 50 percent of the total development time. For many programers, debugging is the hardest part of programming[1].

[1] Da Steve McConnell als Berater für Microsoft tätig war, sollte er wissen wovon er spricht, wenn er von Bugs redet.

1.2 Definition von IEEE [2]

debug (1) To examine or test a procedure, routine, or equipment for the purpose of detecting and correcting errors.

(2) (computing systems). To detect, locate, and remove mistakes from a routine or malfunctions from a computer. See:troubleshoot.

(3) (software) To detect, locate and correct faults in a computer program. Techniques include use of breakpoints[2], desk checking[3], dumps[4], inspection, reversible execution[5], single-step operation[6], and traces[7].

Die Definition von IEEE ist viel allgemeiner als die Definition von McConnell, und sie schließt, im Gegensatz zu ihm, das Testen in das Debugging ein.

Auf meiner Suche nach Definitionen habe ich für Visuelles Debugging leider nichts gefunden, daher eine ad-hoc Definition von mir:

Visuelles Debugging bezeichnet die Anwendung der Techniken der Software Visualisierung auf das Debugging.

2 Debuggingpraxis

Je mehr man sich mit Debugging beschäftigt, sei es aus Forscherlust, sei es eher unfreiwillig, um so mehr fängt man an, sich zu überlegen, wie man die Fehler vermeiden kann. Dazu gehört es, die Fehler zu klassifizieren. Typischerweise macht man beim Programmieren meistens „dumme“ Fehler, und eher selten „gemeine“, aber diese Klassifizierung ist nicht sehr exakt. Es existieren zwar Fehler, die programmiersprachenbedingt häufig auftreten, beispielsweise in der Programmiersprache C Zeigerfehler, aber das wäre auch kein vernünftiger Ansatz für eine Klassifizierung, da man sie für jede Programmiersprache machen müßte. Ich habe mich entschieden, eine Klassifizierung nach der Anstrengung, die notwendig ist, um den Fehler zu erkennen, und zu beseitigen, vorzunehmen.

2.1 Bughierarchie

Die Fehler weiter oben in der Tabelle 1 sind die „Spitze des Eisbergs“ und leicht zu entdecken, da sie sich durch Abstürze oder bei Testläufen bemerkbar machen. Speicherfehler, wie z.B. das Benutzen von bereits freigegebenen Speicher, haben meistens keine so direkten Auswirkungen und sind schwerer zu entdecken. Das gleiche gilt für Synchronisierungs- und Performanzfehler. Bis dahin existieren auch Tools, die dem Programmierer die Fehlersuche mittels Visualisierung

[2] Der Debugger kann das Programm an vorher festgelegten Punkten anhalten, damit man in Ruhe verdächtigen Code inspizieren kann

[3] Manuelles Überprüfen eines Programms mittels Stift, Papier und Schreibtisch

[4] Speicherauszüge, enthalten den Inhalt des Programmspeichers zum Zeitpunkt der Ausgabe des Speicherauszugs, ermöglichen post mortem Debugging, d.h. Debugging eines abgestürzten Programmes

[5] Das Programm auch zu einem früheren Zustand zurückkehren lassen

[6] Die Programmausführung in Einzelschritten verfolgen

[7] Chronologische Auflistungen aufgerufener Funktionen

Implementierungsfehler: Fehler in der Implementation, z.B. Division durch Null

Logische Fehler: Fehler im Algorithmus

Speicherfehler: Fehler beim Speichermanagement/zugriff, z.B. kein Freigeben des allozierten Speichers

Synchronisierungsfehler: Fehler beim Synchronisieren nebenläufiger Prozesse, z.B. Deadlocks

Performanzfehler: Software ist langsamer als sie sein sollte, skaliert nicht gut genug, Cache-unfreundlich

Umgebungsfehler: Der Fehler entsteht durch nicht korrekte Umgebungsparameter (Pfade, Makefiles ...)

Toolfehler: Die Fehler werden von den benutzten Tools eingeführt, z.B. C Compiler, dessen Libraries nicht standardkonform sind

Systemfehler: Fehler werden durch das Betriebssystem oder die Hardwarekomponenten eingeführt, z.B. Bugs in Systembibliotheken oder kaputte Speicherchips

Konzeptfehler: Das Problem liegt im Lösungsvorschlag, z.B. das Problem wurde nicht vollständig analysiert, daher greift auch der Lösungsvorschlag zu kurz

Tabelle 1: Bughierarchie

erleichtern. Bei den anderen Fehlern hilft einem meistens nur der eigene detektivische Spürsinn.

2.2 Debuggingprozeß

Man kann Debugging als einen Prozeß der Informationsgewinnung betrachten, bei dem es darum geht, das Wissen, das man über einen Fehler hat, zu vermehren. Typischerweise macht man dabei sechs Schritte, vom Erkennen des Fehlers bis zum Überprüfen der Lösung, wie in der Tabelle 2 dargestellt.

1.	Erkennen des Fehlers
2.	Stabilisieren des Fehlers
3.	Entwurf von Hypothesen
4.	Testen von Hypothesen
5.	Beheben des Fehlers
6.	Prüfen, ob der Bugfix erfolgreich war

Tabelle 2: Debuggingprozeß

Für diesen Prozeß werden Daten benötigt, die wiederum visualisiert werden können. Im folgenden einige Beispiele:

Statische Informationen
: z.B. durch Metriken, um auf Codebereiche aufmerksam zu werden, die neu erzeugt, oder besonders fehleranfällig, schlecht dokumentiert etc. sind. Visualisierung z.B. durch farbliche Hervorhebung

Dynamische Informationen[8]

- Datenstrukturen (Visualisierung, eigener Vortrag)
- Prozesse (Visualisierung, eigener Vortrag)
- Kontrollfluß (Visualisierung möglich durch Bäume, Graphen oder Diagramme, Systeme nachfolgend)

Speicherinformation

- Lifetime von Objekten
- Speichermanagement
- Cache-Verhalten

Performanz

- Trace-Information
- Profiling

3 Systeme

Visualisierung in Debuggern funktioniert durch das Einfügen von „hook-points“ in das Programm, durch die der Visualisierungscode aufgerufen wird. Man kann die Systeme grob unterteilen in Code-intrusive und Data-intrusive, je nachdem ob ein System die „Hooks“ als spezielle Funktionsaufrufe im Code einfügt, oder die „Hooks“ als neue Methoden in Objektklassen definiert werden, und beim Objektzugriff ausgeführt werden. Außerdem existiert die Unterscheidung zwischen manuellen und automatischen Systemen, wobei bei manuellen Systemen der Programmierer den Visualisierungscode selbst per Hand einfügen (und später ggf. entfernen muß), oder dieser automatisch generiert wird.

3.1 VCC - Visual C Compiler

VCC wurde geschrieben von Ricardo Baeza-Yates, Gaston Quezada, und G. Velmadre [4] von der Universidad de Chile in Santiago. Das System ist am

[8] Durch die Dynamik des Programmablaufs bietet sich Animieren der Informationen an, statt statischer Anzeige

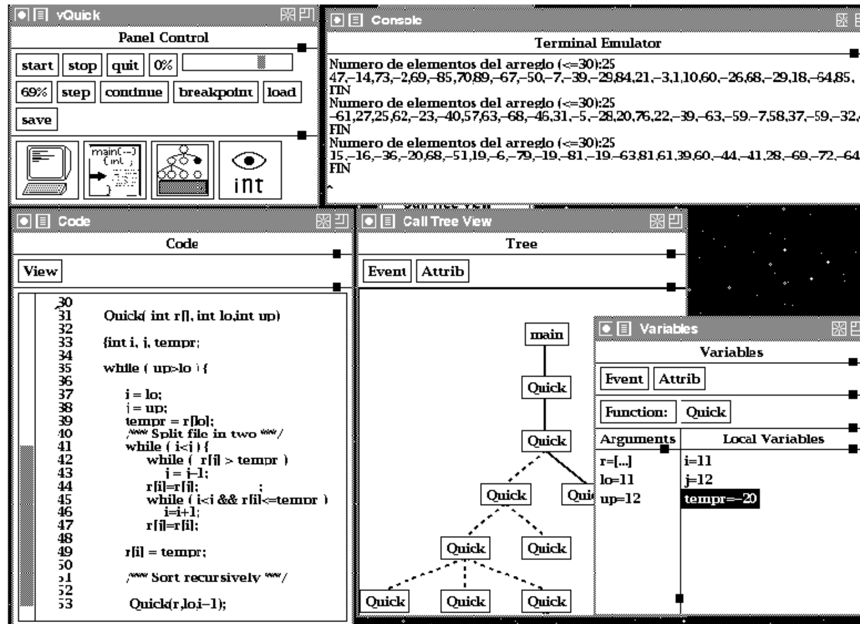


Abbildung 1: Die verschiedenen Sichten des VCC

Anfang der 90er Jahre entstanden. Man kann es über das Internet herunterladen und ausprobieren [3], da das Programm jedoch nur in der ausführbaren Form verfügbar ist, läßt es sich nur auf bestimmten Solaris Versionen mit X Windows ausführen.

VCC visualisiert Programme in der Programmiersprache C, indem es code-intrusive und automatisch Animationen in den Programmcode hinein generiert. Bei der Ausführung bettet VCC das Programm in eine visuelle Debuggingumgebung ein. Es bietet Sichten für Code, aktive Funktion, Funktionsaufrufbaum, sowie Datenstrukturen.

Die einzelnen Sichten von VCC kann man in der Abbildung 1 betrachten. Links oben sieht man das Kontrollfenster mit den üblichen Debuggerkommandos, sowie den Befehlen für das Aufrufen der einzelnen Sichten. Links unten ist das Codefenster, in dem der Programmcode angezeigt wird. Rechts oben befindet sich das Terminalfenster, darunter sind der Funktionsaufrufbaum und die Variablensicht.

Die Implementierung ist ein Beispiel für code-intrusive automatische Systeme. In der Abbildung 2 sieht man wie der Originalquelltext Schritt für Schritt durch Visualisierungsbefehle ergänzt wird, bis zum Schluß das lauffähige Programm entsteht.

Zusammenfassend kann man bei VCC die folgenden Vorteile und Nachteile gegenüberstellen:

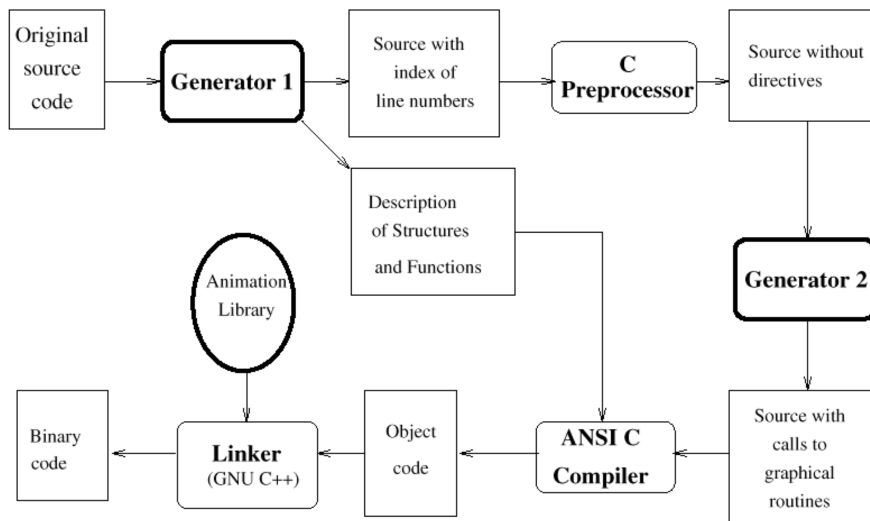


Abbildung 2: Die Implementation des VCC

Vorteile	Nachteile
• automatisch • interaktiv	• auf niedrigem semantischem Niveau • setzt strukturiertes Programmieren voraus

3.2 ZStep95 - A Reversible Animated Source Code Stepper

ZStep95 wurde veröffentlicht von Henry Liebermann und Christopher Fry [5] vom MIT als Weiterentwicklung des Lisp Debuggingumgebung ZStep94, deren Ahnen in die Mitte der 80er Jahre zurückreichen. Geschrieben wurde das System in der Programmiersprache Macintosh Common Lisp 2.0 für eine Teilmenge von Common Lisp, und läuft nur auf Macintosh Rechnern. Weder der Quelltext, noch das ausführbare Programm sind im Internet erhältlich. Es ist vom Typ her, wie VCC, code-intrusive und automatisch.

ZStep95 generiert automatisch eine Animation des Programmablaufs, sowie Sichten für aktuelle Daten, und speichert die History eines Programmlaufs. Man kann dann die History mittels eines „Videorekorders“ vor und zurückspulen, und damit einen Fehler, wie seine Begleitumstände effizienter lokalisieren. Die Entwickler von ZStep95 haben großen Wert auf Benutzerfreundlichkeit und ausführliche Fehlermeldungen gelegt, um damit „Programming by Error Message“ zu ermöglichen. Programming by Error Message ist die Idee, durch intelligente und aussagekräftige Fehlermeldungen dem Programmierer beim Debugging auf die Sprünge zu helfen. Es ist nicht unumstritten, beispielsweise sagt Harlan D. Mills

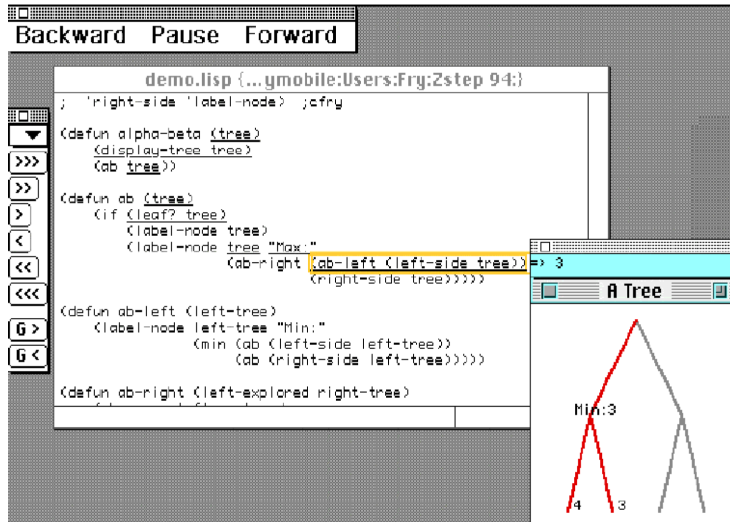


Abbildung 3: Das Interface des ZStep95

[6] zum Thema Programming by Error Message:

> An interactive debugger is an outstanding example of what is not needed - it encourages trial-and-error hacking, rather than systematic design, and it also hides marginal people barely qualified for precision programming.

Andererseits muß man nicht aus der Möglichkeit, daß manche Programmierer interaktive Debugger für ihre Hausaufgaben „mißbrauchen“ sofort schlußfolgern, daß *alle* Programmierer das auch tun.

In der Abbildung 3 sieht man das Interface von ZStep95. Oben und rechts befinden sich die Steuerleisten, die einem Videorecorder nachempfunden sind. Auf den Steuerleisten befinden sich Befehle zum (schnellen) Vor- und Rücklauf, Pausieren, sowie Vor- und Rücklauf zum nächsten visualisierten Ereignis. Das Programm selbst befindet sich in einem Editor in der Mitte der Abbildung 3, in den ZStep95 eingebettet ist, und das Ergebnis der Visualisierung sieht man rechts unten.

Zusammenfassend kann man bei ZStep95 die folgenden Vorteile und Nachteile gegenüberstellen:

Vorteile	Nachteile
• automatisch • interaktiv • benutzerfreundliche Debuggingumgebung • History • Programming by Error Message	• unvollständige Common Lisp Implementierung • wenige, fixe Fehlermeldungen • History benötigt viel Speicherplatz • Programming by Error Message

ZStep95 ist durch die besonders betonte Benutzerfreundlichkeit auf der einen, und die unvollständige Unterstützung von Common Lisp auf der anderen Seite, eher als Lernumgebung denn als Entwicklungsumgebung für größere Projekte geeignet.

3.3 Lens

Lens wurde geschrieben von John T. Stasko und Sougata Mukherjea vom Georgia Institute of Technology [7] in C für UNIX Betriebssysteme mit X Windows, der graphischen Bibliothek Motif und dem Debugger dbx. Lens ist frei erhältlich über das Internet [8] inklusive des Quelltextes, so daß es auf den meisten UNIX-Plattformen übersetzt und benutzt werden kann.

In der weiter oben definiert Klassifizierung läßt sich Lens als ein manuelles, Code-intrusive System beschreiben. Es ist *kein* Debugger an sich, sondern ein Frontend für das Visualisierungssystem XTango [9, 10] und den Debugger dbx. Es erlaubt über ein visuelles Designtool interaktiv Animationen zum Code zu erstellen. Dabei wurden (eine Lehre aus der Entwicklung und Benutzung von XTango) die graphischen Ausdrucksmöglichkeiten auf wenige Primitive reduziert. Man kann die begleitenden Animationen eines Programms speichern, so daß das Debugging später wiederaufgenommen werden kann.

In der Abbildung 4 sieht man Lens in Aktion: Im Vordergrund befindet sich das Animationsfenster von XTango, während man im Hintergrund das Hauptfenster von Lens sehen kann. Darin befinden sich links der Quelltext des Programmes[9], sowie rechts der Animationseditor und das Ausgabefenster des Debuggers.

Zusammenfassend kann man bei Lens die folgenden Vorteile und Nachteile gegenüberstellen:

[9]Die kleinen α-s am linken Rand signalisieren, daß zu den Befehlen eine Visualisierung existiert

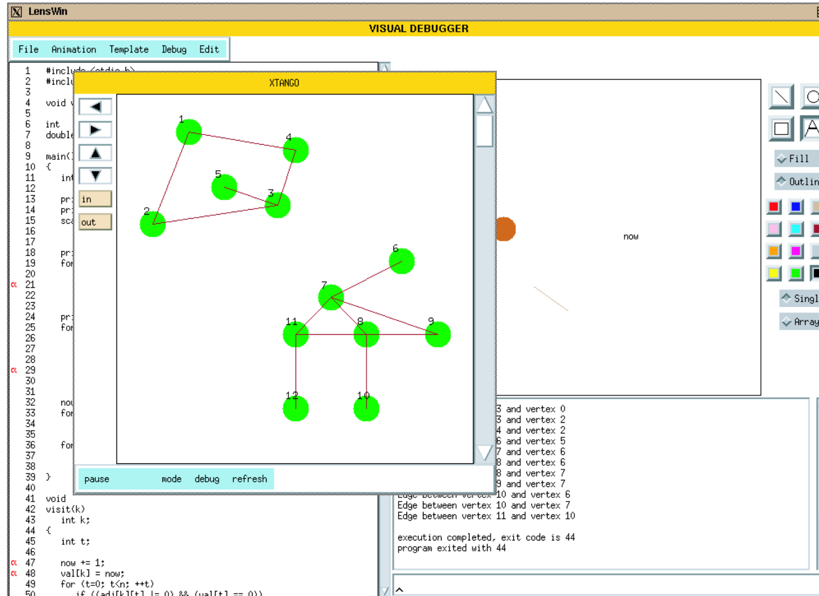


Abbildung 4: Das Interface von Lens

Vorteile	Nachteile
• interaktiv • hoher Freiheitsgrad • vielseitig verwendbar (z.B. Codedokumentation) • einfach zu erlernen/benutzen • benutzt echten Debugger	• manuell • keine Kontrollstrukturen für Animationen • Debuggerabhängig • zeitintensiv

3.4 DDD - Data Display Debugger

DDD [11] wurde Mitte der 90er Jahre entwickelt von Andreas Zeller und Dorothea Krabiell [12] von der TU Braunschweig, und wird fortwährend weiterentwickelt. Dieses Programm ist ebenfalls ein Debuggerfrontend. Da DDD ein Debuggerfrontend für den frei erhältlichen und weit verbreiteten GNU Debugger gdb ist, profitiert er von dessen Weiterentwicklung. DDD kann zum Debugging von C,C++, Fortran, Assembler, Python, Perl und Java benutzt werden, wobei er bei manchen Programmiersprachen auch als Frontend für die Standarddebugger, z.B. jdb in Java funktioniert. Das Programm benötigt zum Betrieb X Windows und ein graphisches Toolkit, entweder Motif oder dessen freien Klon

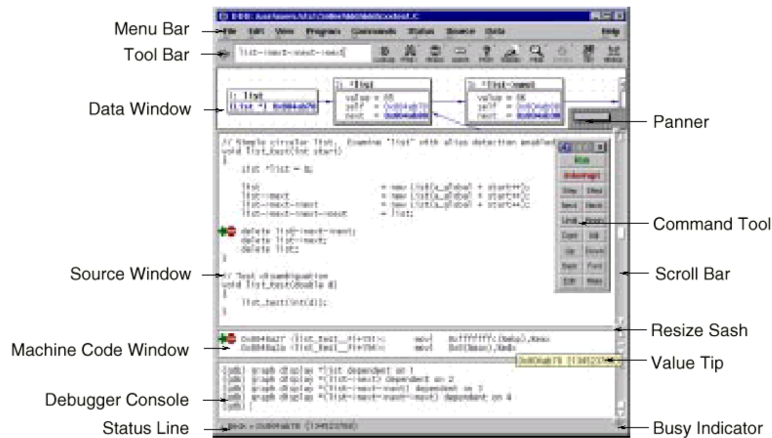


Abbildung 5: Das Hauptfenster von DDD

LessTiff. Es existieren bereits vorcompillierte Versionen für die meisten UNIX-Betriebssysteme, sowie für Microsoft Windows.

DDD ist in erster Linie *kein* Visualisierungstool an sich, sondern benutzt zur Visualisierung externe Programme wie gnuplot, vereinigt diese aber mit den Debuggern unter einer komfortablen und leicht erlernbaren Oberfläche. Durch die Benutzung externer Tools schaffen sich die Entwickler eine Menge Arbeit vom Hals, um automatisch von der weiteren Entwicklung dieser externer Tools zu profitieren. Es enthält Sichten für den Quelltext, Maschinencode, Terminal- bzw. Debuggerausgabe, und Daten, die man in den Abbildungen 5, 6 und 7 betrachten kann. Im Quelltext lassen sich beispielsweise Breakpoints mit zwei Mausklicks setzen, verändern oder löschen. In der Datensicht lassen sich die Daten anzeigen, Zeiger verfolgen, und größere Datengraphen automatisch layouten. Im Terminal kann man wie gewohnt mit dem Debugger kommunizieren, indem man Befehle eintippt. Weitaus mehr vergnügen bereitet es jedoch diese über das Benutzerinterface zu erteilen.

In der Abbildung 8 ist beispielsweise der Breakpointeditor dargestellt, der es einem sogar erlaubt DDD-Makros aufzunehmen (beispielsweise um Daten automatisch von gnuplot oder einem anderen Tool visualisieren zu lassen), Bedingungen für das Anhalten des Programms zu setzen und sogar die Anzahl der zu ignorierenden Treffer, was sich vor allem in Schleifen als nützlich erweist. Der Breakpointeditor macht es auch Programmierern, die noch nicht in die Tiefen des benutzten Debuggers eingetaucht sind, sehr einfach, dessen komplexe Features zu nutzen, um ihre Produktivität zu steigern.

Bei der Datensicht sieht man in der Abbildung 9 eine dreielementige Liste, deren letztes Element wieder auf das erste Element zeigt. DDD erkennt bei der Visualisierung der Datenstruktur, daß es sich beim Nachfolger des letzten Elements um das erste Element handelt, und stellt das automatisch durch einen Pfeil zum ersten Element dar, wenn der Benutzer den Nachfolger sich anzeigen lassen möchte. Mit farbiger Schrift wird signalisiert, daß es sich bei dem Datum um einen Zeiger handelt. Ist die Schrift farbig *und* fett, so wurde der Zeiger bereits weiterverfolgt, sonst kann man sich mit einem Mausklick das Element

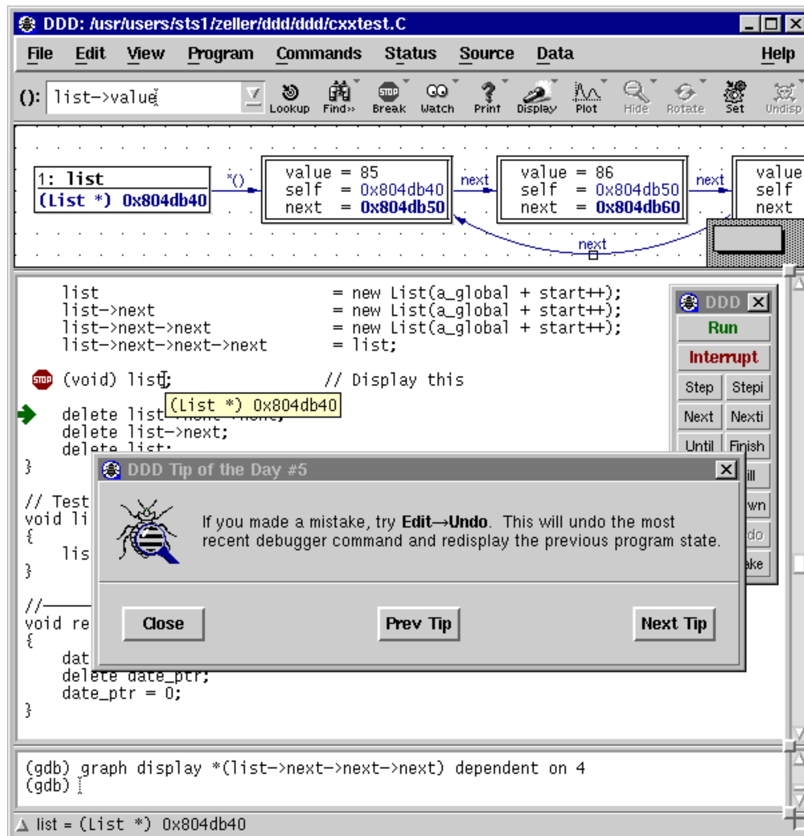


Abbildung 6: DDD in Aktion

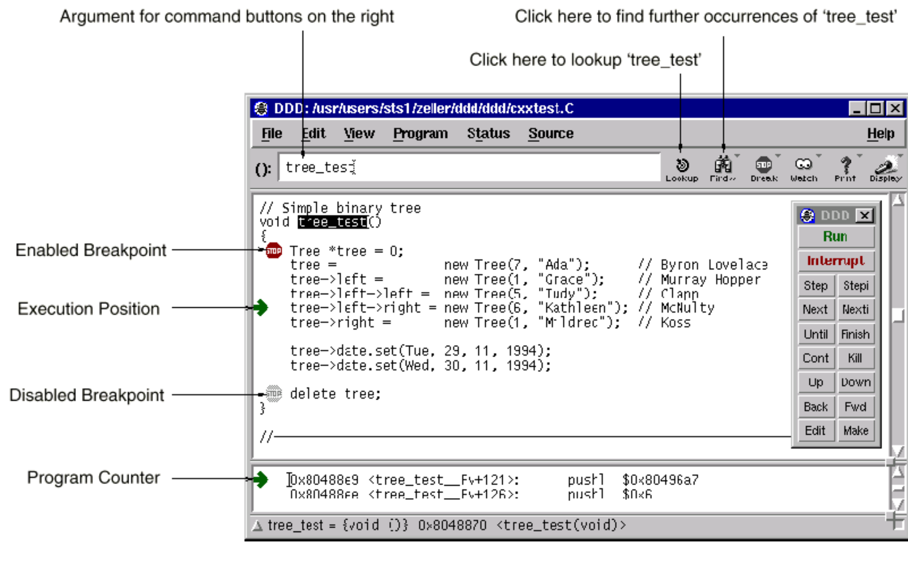


Abbildung 7: Quelltextfenster des DDD

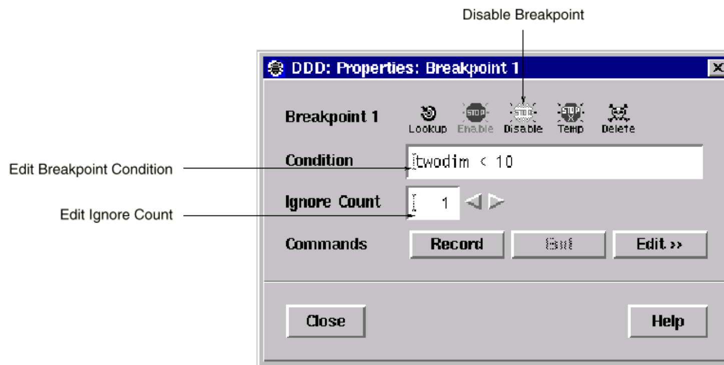
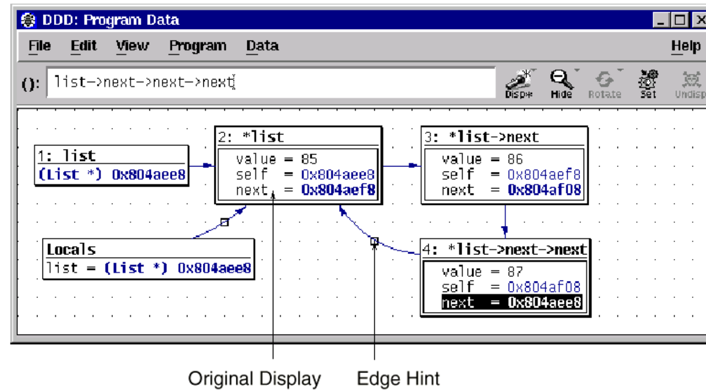


Abbildung 8: Der Breakpointeditor des DDD



Examining Shared Data Structures

Abbildung 9: Die Datensicht des DDD

anzeigen lassen, auf das der Zeiger zeigt. Das alles geschieht automatisch, ohne eine Zeile Visualisierungscode schreiben oder designen zu müssen, und auch ohne hook-points im Programm zu definieren und das Programm neu übersetzen zu müssen. Das könnte man scherzhaft „visualization on demand“ nennen.

Eine der Stärken von bisher besprochenen Debuggern ist die Visualisierung von Arrays, beispielsweise als zweidimensionale Balkengraphiken. DDD in Verbindung mit gnuplot kann das auch, und noch viel mehr: Da DDD auch eine Programm-History speichert, lassen sich die Inhalte eines Arrays über einen Zeitraum dreidimensional darstellen, wie in Abbildung 10. Durch die Flexibilität der beiden Tools sind noch viele weiteren Spielarten möglich.

Zusammenfassend kann man bei DDD die folgenden Vorteile und Nachteile gegenüberstellen:

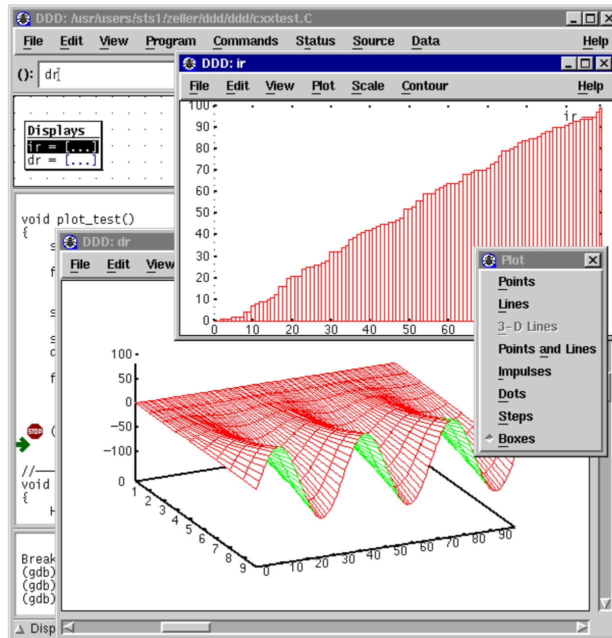


Abbildung 10: Das Zusammenspiel von DDD und gnuplot

Vorteile	Nachteile
• interaktiv	• keine Animationen
• einfach zu erlernen	• kein Call-Tree
• einfach zu benutzen	• keine Visualisierung von Profilingdaten
• benutzt echte Debugger	
• History	
• vielfältige Datensichten	
• remote Debugging	
• Speichern und Laden von Sitzungen	
• kann in andere Programme eingebunden werden	
• vitales Open-Source-Projekt	

4 Kollaboratives Debugging

Es gibt Fehler mit denen man Stunden, manchmal auch Tage verbringt, ohne sie loszuwerden. Frustriert zeigt man dann den Code einem Kollegen, der nach ein paar Augenblicken mit einem Vorschlag kommt, an den man selbst nie gedacht hätte, und den Fehler behebt. Da der andere Programmierer vielleicht mal selbst in einer solchen Lage war, war es für ihn einfach einem zu helfen. Das ist der Vorteil des „peer-reviews" : man nutzt das Expertenwissen der *anderen* Programmierer.

Das klappt ganz gut, solange sich das Problem vernünftig in wenigen Sätzen umreißen läßt. Wenn man jedoch den Bug nur sehr schwach einkreisen kann, dann hilft dem Kollegen nur noch der Blick in den Quelltext[10], und der kann sehr anstrengend sein. Daher sieht man auch in den einschlägigen Newsgroups wie comp.lang.c zwar viele Antworten auf typische Anfängerfragen[11], aber nur wenige Analysen vertrackter Problemstellungen, obwohl heutezutage immer ein Experte online ist. Das liegt daran, daß es eben sehr anstrengend sein kann. Wünschenswert wäre deshalb eine Methode um möglichst exakt einen Bug und das gewünschte Verhalten zu beschreiben, und mittels einer Programmhistory reproduzierbar zu machen, damit die Fehlersuche in einem fremden Programm einfacher wird.

4.1 Internet Software Visualization Library

Die ISVL wurde entworfen von John Domingue und Paul Mulholland [13, 14] von der Open University in Milton Keynes. Das System basiert auf einer Client-Server Architektur, die mittels Java und HTML Programme auf jedem Java-fähigen Browser visualisieren kann. Durch die Verwendung von Standardsprachen im Internet macht man sich die rasante Entwicklung der letzten Jahre zu nutze, deren letztes Ergebnis Webbrowser auf Handys sind.[12] Das System unterstützt das Debuggen von Prolog Programmen.

Der ISVL Server in der Abbildung 11 besteht aus drei Teilen: einem angepaßten Webserver, auf dem die allgemeine Visualisierungskomponente sitzt, und schließlich die programmiersprachenspezifische Visualisierungskomponente. Die History des Programms wird mitsamt etwaigen Anmerkungen des Programmierers in einem „Movie" gespeichert, und auf den Server geladen. Man sieht das annotierte Programm in der Abbildung 12. Oben befindet sich das Ausgabefenster, in dem beispielsweise der Trace des Programmes ausgegeben wird. In der Mitte befindet sich die Visualisierung des Programmes, sowie die entsprechenden Anmerkungen des Programmierers. Das ist die Arbeitsfläche des Programms. Unten findet man die (an die Videorekordermetapher angelehnte) Kontrolloberfläche des ISVL Clients, mit dem man Visualisierungen aufzeichnen, versenden, abspielen und synchronisieren kann. Eine Antwort auf die obengenannte Fehlerbeschreibung sieht man in der Abbildung 13.

Durch seine Portabilität und die Möglichkeit, auch auf billigen Plattformen zu laufen, eignet sich ISVL hervorragend für den Einsatz im Netz; durch die kompakten Movies, und die intelligente Client-Server Architektur wird die Netzlast

[10] Der Blick in den Quelltext ist jedoch nicht immer möglich, z.B. aus Urheberrechtsgründen

[11] Die meisten verweisen dabei auf die FAQs der jeweiligen Newsgruppe

[12] Eine so angenehme Vorstellung, Programme auf Handydisplays zu visualisieren und zu debuggen ist das nun auch wieder nicht

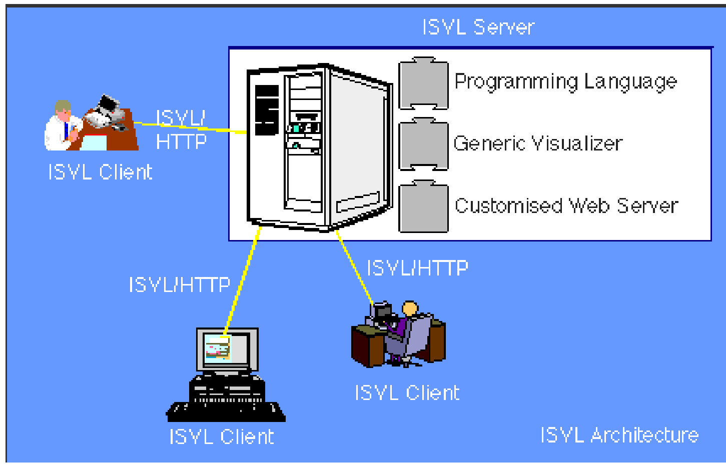


Abbildung 11: Die Client-Server Architektur der ISVL

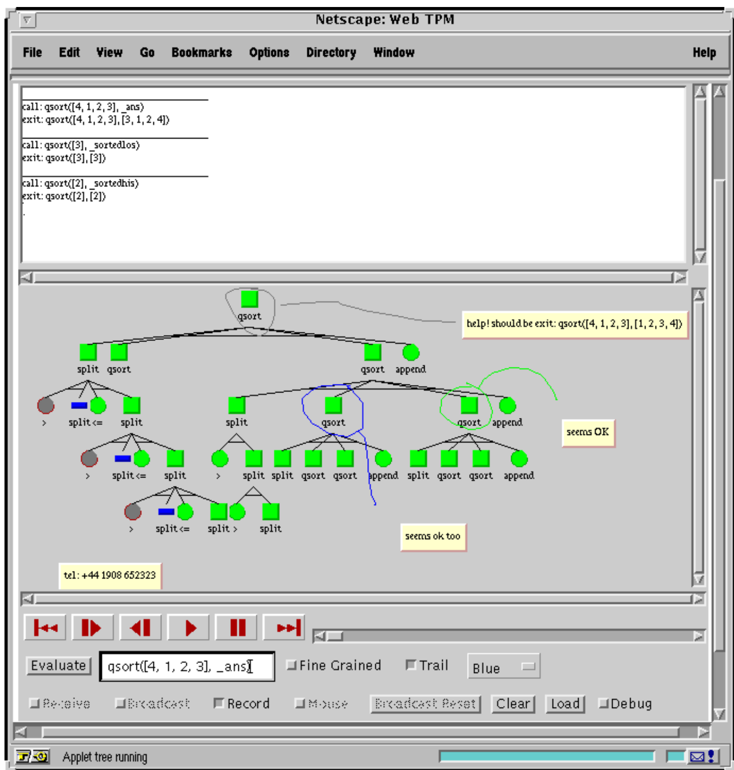


Abbildung 12: Eine Fehlerbeschreibung in ISVL

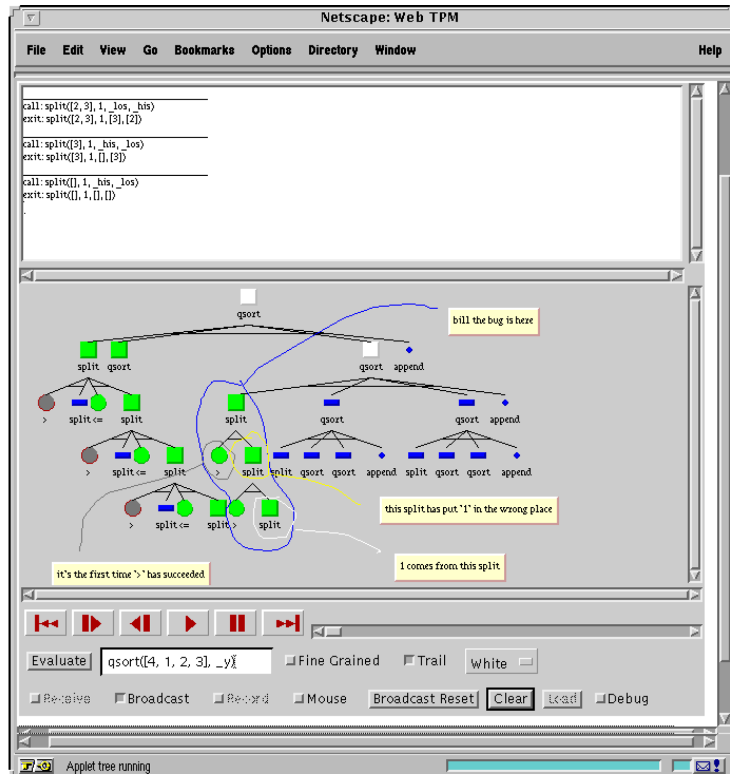


Abbildung 13: ISVL in Aktion

auf ein verträgliches Niveau reduziert; durch die History und verschiedene Visualisierungsmöglichkeiten wird Debugging nicht-trivialer Programme auf eine komfortable Art und Weise möglich: ISVL überzeugt durch ihre Vorteile. Der einzige Nachteil ist die noch fehlende Unterstützung für das Debugging von weit verbreiteten Programmiersprachen wie Java oder C++.

5 Zusammenfassung und Ausblick

Ich habe in meinem Vortrag versucht einen kleinen Überblick über die Fortschritte bei visuellen Debuggern in den letzten zehn Jahren zu geben. Gegenüber den alten Textinterfaces haben visuelle Debugger den Vorteil der intuitiveren Bedienung, der, wenn die Verbindung mit den Techniken der Software Visualisierung gelingt, die Arbeit mit dem Debugger deutlich angenehmer und effizienter macht, wie beim DDD. Weitere Fortschritte sind die automatische Erstellung der History im Debugger, und deren Speichern, um eine Debuggingsitzung fortzuführen, oder Programme quelltextunabhängig zu visualisieren, wie in der ISVL.

Der Erfolg von Open Source Projekten, und der dadurch bedingte Erfolg des Internets als Entwicklungsplatform führen zum Entstehen von Entwicklergemeinschaften. Diese scheinen der Trend der Zukunft zu sein, und dort werden weitere interessante Tools entstehen, um das gemeinsame Arbeiten einfacher zu machen, auch mittels Software Visualisierung. Versuche die Schnittstellen von verschiedenen Debuggern zu standardisieren, wie der des High Performance Debugging Forums [15], werden zu zunehmenden Arbeitsteilung wie im DDD führen, wobei der Textdebugger unter dem visuellen Debugger eher an Bedeutung verlieren wird. Dadurch wird es deutlich einfacher eine breite Palette an neuen Programmiersprachen zu debuggen.

Debugging ist wieder interessanter geworden.

Literatur

[1] S. McConnell: Code Complete, Microsoft Press (1993).

[2] The New IEEE Standard Dictionary of Electrical and Electronic Terms, Fifth Edition, IEEE (1993).

[3] ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/vcc/

[4] R.A. Baeza-Yates, G. Quezada, G. Valmadre: Visual Debugging And Automatic Animation Of C Programs, in Peter Eades and Kang Zhang, editors: Software Visualisation, in Shi-Kuo Chang's Software Engineering and Knowledge Engineering Book Series. World Scientific Press (1996).

ftp://sunsite.dcc.uchile.cl/pub/users/rbaeza/vcc/vcc.ps.gz

[5] H. Liebermann, C. Fry: ZStep95: A Reversible, Animated Source Code Stepper, in Software Visualization: Programming as a Multimedia Experience, John Stasko, John Domingue, Blaine Price, Marc Brown, eds., MIT Press (1997).

http://lieber.www.media.mit.edu/people/lieber/Lieberary/ZStep/ZStep-SoftViz/ZStep-SoftViz.html

[6] S. 646 in S. McConnell: Code Complete, Microsoft Press (1993).

[7] J. T. Stasko, S. Mukherjea: Toward Visual Debugging: Integrating Algorithm Animation Capabilities Within a Source-Level Debugger, ACM Transactions on Computer-Human Interaction, Vol. 1, No.3, September 1994, S. 215-244

[8] ftp://ftp.cc.gatech.edu/pub/people/stasko/lens.tar.Z

[9] J. T. Stasko: Animating algorithms with XTANGO, SIGACT News 23, 2 (Spring), S. 67-71

[10] ftp://ftp.cc.gatech.edu/pub/people/stasko/xtango.tar.Z

[11] http://www.cs.tu-bs.de/softech/ddd/

[12] http://www.cs.tu-bs.de/softech/ddd/ftp/doc/man/ddd.man.ps.gz

[13] J. Domingue, P. Mulholland: Fostering Debugging Communities On The Web, Communications of the ACM, Vol 40, No. 4, S. 65-71.

ftp://frost.open.ac.uk/pub/paulm/CACM97.ps.gz

[14] http://kmi.open.ac.uk/~paulm/isvl.html

[15] http://www.ptools.org/hpdf/