Visuelle Programmierung - Potential und Grenzen

Stefan Schiffer [*)]

Visuelle Programmierung ist zu einem Begriff geworden, der für intuitive Softwareentwicklung steht. Bei der visuellen Programmierung verzichtet man weitgehend auf textuelle Notationen und verwendet statt dessen graphische Programmbausteine. Mit dem Einsatz visueller Techniken ist unter anderem die Hoffnung verbunden, Programme einfacher und besser als bisher erstellen und verstehen zu können, womit auch Programmierlaien der Zugang zur Softwareentwicklung geöffnet würde. Dies könnte zu einem Abbau des vielerorts enormen Anwendungsrückstaus führen. Den Vorteilen von visueller Programmierung wie anschaulicher Realitätsbezug, großes Motivations- und Lernpotential, Abschwächung syntaktischer Strukturen und Betonung semantischer Zusammenhänge stehen Nachteile wie fehlende Standards, geringe Darstellungsdichte, hohe Resistenz gegenüber Modifikationen, schwierige Formalisierbarkeit und beschränkte Abstraktionsmöglichkeiten gegenüber. Der Artikel lotet Potential und Grenzen der visuellen Programmierung aus und versucht eine Orientierungshilfe bei der Einschätzung dieser neuen Programmiertechnik zu geben.

1. Einleitung

Visuelle Programmierung (VP) hat in den letzten Jahren stark an Popularität gewonnen, vor allem wegen des Versprechens, daß diese neue Art der Programmierung die Softwareentwicklung bedeutend vereinfacht. Graphische Entwicklungsumgebungen sollen sowohl professionelle Programmierer als auch Anwender beim Einsatz des Computers durch intuitive Konzepte unterstützen. Befürworter der VP meinen, daß dadurch ein beträchtlicher Teil der Anwendungsentwicklung vom Softwareingenieur zum Anwender verlagert würde. Der Benutzer soll mit VP maßgeschneiderte Applikationen erstellen oder vorhandene Software an seine Vorstellungen anpassen können, ohne dafür besondere Programmierkenntnisse zu benötigen. Dadurch könnte der vielerorts enorme

[*)] C. Doppler Labor für Software Engineering, Johannes Kepler Universität Linz, A-4040 Linz, Austria

Anwendungsrückstau abgebaut werden. Diese Erwartungen erinnern an Hoffnungen, die man vor mehr als 30 Jahren mit höheren Programmiersprachen verband. Man meinte damals, daß diese Sprachen den Programmierer überflüssig machen könnten, weil ihre starke Anlehnung an Englisch jeden Sprach- und Sachkundigen in die Lage versetzen würde, mit Datenverarbeitungsanlagen ähnlich wie mit Arbeitskollegen zu kommunizieren. Wir wissen, daß sich diese Einschätzung als Irrtum erwies: tatsächlich blieb die Softwareentwicklung trotz höherer Programmiersprachen in der Hand von Spezialisten und der Bedarf an Programmierer wurde größer statt kleiner. VP wird ebenfalls keine Arbeitslosigkeit unter Softwareentwicklern verursachen, denn die Programmierung zuverlässiger und effizienter Softwaresysteme verlangt neben guten Werkzeugen auch immer die Kenntnis softwaretechnischer Methoden, die nicht mit dem Kauf einer VP-Umgebung erworben werden können.

Die Softwareindustrie geht dennoch von hohen Umsätzen für VP-Umgebungen aus. Snell [24] zitiert eine Studie, die VP-Produkten für Ende 1999 einen Marktanteil von 3,8 Milliarden USD voraussagt. Die höchsten Chancen werden Werkzeugen eingeräumt, die zur Entwicklung von Client-Server-Anwendungen geeignet sind. 30 % aller Unternehmen, die solche Werkzeuge verwenden, setzen laut dieser Studie bereits heute auf die VP-Umgebung *Visual Basic*. Eine wichtige Rolle wird VP auch bei Integration von Datenbankwerkzeugen in Software-Entwicklungsumgebungen zugesprochen. Das Zusammenrücken von Datenbankanbietern und Herstellern von Software-Entwicklungsumgebungen, um gemeinsam visuelle Werkzeuge zu entwickeln, ist ein deutliches Signal in diese Richtung.

In Folge des hohen Marktpotentials werden eine Vielzahl neuer Programmierumgebungen angeboten, die das Etikett "visuell" tragen. Als Beispiele für das breite Spektrum kommerzieller Produkte seien *VisualWorks* von ParcPlace und *LabVIEW* von National Instruments genannt. VisualWorks ist eine erweiterte Version der objektorientierten Entwicklungsumgebung ObjectWorks. Der Unterschied zwischen ObjectWorks und VisualWorks besteht vor allem darin, daß in ObjectWorks Benutzungsoberflächen durch textuelle Beschreibungen definiert werden, während in VisualWorks dazu ein graphischer Benutzungsschnittstelleneditor verwendet wird. Die Applikationen werden in beiden Fällen in Smalltalk geschrieben; eine visuelle Programmiersprache kommt nicht zum Einsatz.

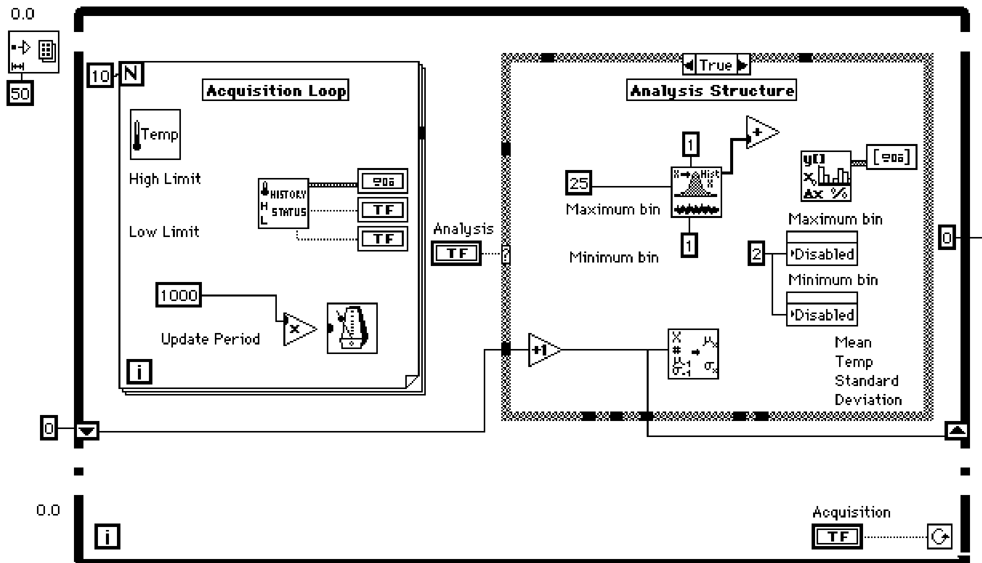


Abbildung 1: Ein LabVIEW-Programm zur Temperaturanalyse.

Anders sieht es bei LabVIEW aus, einer Programmierumgebung für den Bau von virtuellen Instrumenten. In LabVIEW wird sowohl die Benutzungsschnittstelle als auch die Programmlogik mit graphischen Bausteinen erstellt. Text spielt nur eine geringe Rolle und wird vor allem für Beschriftungen, Kommentare sowie Zahlen- und Zeichenkonstanten verwendet. Abbildung 1 zeigt ein typisches LabVIEW-Programm.

Klassifiziert man VP-Systeme anhand des Kriteriums “Verfügbarkeit einer visuellen Programmiersprache”, so ergeben sich zwei Gruppen kommerzieller Systeme (exemplarische Auswahl, nicht alle verfügbaren Systeme sind aufgezählt):

- Visuelle Programmiersysteme mit *visueller* (graphischer) Programmiersprache: *AVS* von Advanced Visual Systems, *LabVIEW* von National Instruments, *Parts* von Digitalk, *Prograph* von Pictorius Inc., *VisualAge* von IBM, *VEE* von HP. Bei diesen Umgebungen enthält der Programmcode einen hohen Anteil an graphischen Elementen und nur wenig Text.

- Visuelle Programmiersysteme mit *verbaler* (textueller) Programmiersprache: *VisualWorks* von ParcPlace, *PowerBuilder* von Powersoft sowie *Visual Basic* und *Visual C++* von Microsoft. Bei diesen Umgebungen kommen graphische Komponenten meist nur beim Bau der Benutzungsschnittstelle zum Einsatz, während zur Darstellung der Programmlogik eine textuelle Programmiersprache verwendet wird.

Dieser Beitrag konzentriert sich vor allem auf die Eigenschaften visueller Notationen zur Definition von Programmen, d.h. auf Systeme der ersten Kategorie. Nur mit solchen Systemen, die eine visuelle Programmiersprache unterstützen, ist VP im engeren Sinn möglich (vgl. Chang [3] und Myers [17]). Eine Auseinandersetzung mit den Vor- und Nachteilen dieser Systeme ist deshalb von besonderem Interesse, weil wissenschaftliche Arbeiten und kommerzielle Produktankündigungen üblicherweise nur die Stärken visueller Ansätze herausstreichen. Kritische Anmerkungen finden sich nur selten, sind aber notwendig, um zu einer realistischen Einschätzung zu gelangen. VP-Systeme mit textueller Programmiersprache und graphischem Benutzungsschnittstelleneditor werden nur ansatzweise diskutiert, weil über den hohen Wert visueller Werkzeuge bei der Gestaltung von Benutzungsoberflächen allgemeiner Konsens herrscht. Von VP abzugrenzen sind zudem Systeme zur Softwarevisualisierung: während bei VP die Implementierung im Vordergrund steht, geht es bei Softwarevisualisierung um die verständliche Illustration softwaretechnischer Sachverhalte mit visuellen Hilfsmitteln, unabhängig davon, ob die zugrundeliegenden Beschreibungsmittel verbaler oder visueller Natur sind. Weil dieser Beitrag vor allem auf die Konstruktion von Softwaresystemen mit visuellen Mitteln eingeht, wird Softwarevisualisierung nur kurz angesprochen.

2. Begriffliche Einführung

Die erste Hürde, die jeder Definitionsversuch für VP überwinden muß, ist der Begriff “visuell”, der im allgemeinen “das Sehen oder den Gesichtssinn betreffend” bedeutet und somit eine bestimmte Wahrnehmungsart bezeichnet. Die Eigenschaft “visuell” bezieht sich nicht nur auf graphische Elemente, sondern auch auf gewöhnlichen Text, der ebenso sichtbar ist, wie etwa ein Piktogramm. Der Komplementärbegriff zu “visuell” ist demnach der Begriff “verbal” und nicht - wie oft unbedacht verwendet - der Begriff “textuell”. Die Differenzierung zwischen “visuell” und “verbal” deckt sich auch mit der Begriffswelt der Kognitionspsychologie, die zwischen der Verarbeitung verbaler und visueller Informationen unterscheidet. Im Rahmen dieses Beitrags ist unter “visuell” jene Eigenschaft von Objekten zu verstehen, die zur Folge hat, daß signifikante Informationen über ein Objekt nur über das visuelle Wahrnehmungssystem des Menschen erfaßbar sind. Mit anderen Worten: ein Objekt weist dann die Eigenschaft “visuell” auf, wenn mindestens eine Information über das Objekt, die für die Durchführung einer bestimmten Aktivität unverzichtbar ist, über andere Sinne nicht gewonnen werden kann. Visuell ist somit als “visuell informativ” zu verstehen. Rein

textuelle Objekte sind zwar sichtbar, aber nicht visuell informativ, weil man die dargestellten Informationen nicht nur über das Auge, sondern ohne Informationsverlust auch über das Gehör und andere Sinne aufnehmen kann. Farbige Objekte hingegen sind visuell informativ (falls die Farben eine Bedeutung haben), weil die Interpretation der Farben nur über das visuelle Wahrnehmungssystem erfolgen kann.

Visuelle Programmierung ist eine Tätigkeit bei der Software implementiert wird. Die Eigenschaften der dabei verwendeten Materialien (Softwarebeschreibungen und Softwarebausteine) kommen primär visuell zum Ausdruck und die Manipulation der Materialien erfolgt beinahe ausschließlich durch Mechanismen mit visueller Rückkoppelung. Der Begriff “Materialien” soll verdeutlichen, daß sich die visuellen Aspekte nicht auf die eingesetzten Werkzeuge, sondern auf die verwendeten “Baustoffe” beziehen. Werkzeuge, die zwar eine visuelle Schnittstelle aufweisen, aber zur Manipulation von verbalen (textuellen) Materialien bestimmt sind, bilden keine Grundlage für VP. Von VP darf auch dann gesprochen werden, wenn mit den zur Verfügung stehenden Materialien nur sehr einfache Software erstellbar ist. Beispielsweise ist das Ausführen der Aktion “Dokument in einen Ordner legen” auf einem elektronischen Schreibtisch, bereits dann VP, wenn die Aktion mit einen Makrorekorder aufgezeichnet wird und später wiederholbar ist. Ob die visuellen Ausdrucksmöglichkeiten es gestatten, Programme mit Verzweigungen und Wiederholungen zu definieren, ist in diesem Zusammenhang unerheblich.

Eine *visuelle Programmiersprache* ist eine formale Sprache mit visueller Syntax oder visueller Semantik zur vollständigen Beschreibung der Eigenschaften von Software. Syntax und Semantik visueller Programmiersprachen beziehen meist den zwei- oder dreidimensionalen Raum ein, während verbale Programmiersprachen auf eindimensionalen Zeichenketten beruhen. Mit einer *visuellen Universalprogrammiersprache* kann jedes Berechnungsverfahren formuliert werden, das auf einer Turingmaschine ausführbar ist, d.h. eine solche Sprache ist berechnungsuniversell. Eine *visuelle Spezialprogrammiersprache* deckt nur begrenzte Problembereiche ab und ist nicht berechnungsuniversell, weil wesentliche Sprachkonstrukte wie Schleifen oder Verzweigungen fehlen. Für einen eingeschränkten Anwendungsbereich können mit Spezialsprachen dennoch komplette Programme erstellt werden.

3. Reflexionen über fünf Thesen zur visuellen Programmierung

Befürworter der VP zitieren gerne den Satz "Ein Bild sagt mehr als tausend Worte" und gehen davon aus, daß diese im Alltag gewonnene Erfahrung auch für weite Bereiche der Softwareentwicklung gilt. Wenn es um den Wert graphischer Darstellungen in der Programmierung geht, spielen jedoch generelle Überlegungen zu den Vor- und Nachteilen von Bildern eine untergeordnete Rolle. Das Augenmerk muß scharf auf das wesentliche Merkmal der Programmierung gerichtet sein: Programmieren im engeren Sinn ist der Einsatz formaler Mittel zur präzisen und klaren Beschreibung von Anweisungen oder Sachverhalten, die der Computer interpretieren kann. Die dazu verwendeten Notationsformen dürfen keinen Deutungsspielraum erlauben - weder für den Menschen noch für die Maschine. In ihrem aufschlußreichen Artikel "Why Looking Isn't Always Seeing" bringt Petre [19] die Sache auf den Punkt, wenn sie das genußvolle Betrachten eines Kunstwerks der analytischen Interpretation eines Programms gegenüberstellt:

> *The implicit model behind at least some of the claims that graphical representations are superiour to textual ones is that the programmer takes in a program in the same way that a viewer takes in a painting: by standing in front of it and soaking it in, letting the eye wander from place to place, receiving a 'gestalt' impression of the whole. But one purpose of programs is to present information clearly and unambiguously. Effective use requires purposeful perusal, not the unfettered, wandering eye of the casual art viewer. The aim is not poetic interpretation, but reliable interpretation.*

Petre zeigt, daß die Effektivität eines Beschreibungsmittels nicht hauptsächlich davon abhängt, ob es visuelle oder verbale Ausdrucksformen erlaubt, sondern ob die Möglichkeit besteht, durch gute äußere Form zusätzliche Lesehinweise zu geben. Demgemäß gibt es in der Softwareentwicklung zwei Notationsebenen: (1) Die Ebene der *primären Notation*, wo Software mit graphischen oder textuellen Mitteln formal beschrieben wird und (2) die Ebene der *sekundären Notation*, wo informelle Interpretationshilfen für die formalen Ausdrücke der primären Notation gegeben werden. Zur sekundären Notation gehören gestaltende Maßnahmen wie Gruppierungen, Leerraum, Markierungen und Symmetrie. Das Verständnis graphischer Darstellungen hängt stark vom klugen Gebrauch sekundärer Notation ab. Leider sind nur wenige Personen in der Lage, gute Graphiken zu entwerfen. Bei visuellen Programmen besteht aus diesem Grund die Gefahr, oft mit schlecht gestalteten Graphiken konfrontiert zu werden, die auf Ebene der sekundären Notation mehr Verwirrung stiften, als schlecht strukturierte Texte.

Es verwundert deshalb nicht, daß radikale Ansätze aus der Anfangszeit der VP gescheitert sind. Diese ersten Versuche hatten zum Ziel, wenn immer möglich Text aus Programmen zu verbannen, da die eindimensionale, textuelle und statische Repräsentation traditioneller Programmiersprachen als Hauptverursacher für Frustration und Mißerfolg gesehen wurde (vgl. Glinert und Tanimoto [9]). Heute besteht Einigkeit darüber, daß Programmieren mit Bildern alleine nicht praktikabel ist. Es zeigte sich, daß die kompromißlose Beschränkung auf graphische Elemente neben den zuvor angesprochenen ästhetischen Problemen auch die Anwendung fundamentaler Programmierprinzipien vereitelt und dadurch den Programmierprozeß behindert statt ihn zu fördern. Zu diesen Prinzipien gehören etwa die Bezeichnung von Programmkomponenten mit aussagekräftigen Namen oder die Erläuterung wichtiger Programmstellen durch Kommentare. Aktuelle Entwicklungen beschreiten vernünftigere Wege: Text und Graphik ergänzen einander, wobei die Graphik nach wie vor im Mittelpunkt steht.

Die nächsten Abschnitte beschäftigen sich mit angeblichen Vorteilen bildlicher Darstellungen in der Programmierung. Die Diskussion durchleuchtet fünf häufig genannte Thesen und zeigt, daß manche Behauptungen über den hohen Wert von Bildern in der Programmierung zweifelhaft sind. Grundlage der präsentierten Schlußfolgerungen sind Literaturstudien sowie Erfahrungen und Beobachtungen anhand einer Reihe von universitären Programmierprojekten mit kommerziell verfügbaren VP-Umgebungen. Zudem hat der Autor selbst eine umfangreiche multiparadigmen-orientierte VP-Umgebung entworfen und implementiert (vgl. Schiffer und Fröhlich [22]). Die dabei gewonnenen Erkenntnisse flossen ebenfalls in diesen Aufsatz ein.

3.1 These: Bilder sind mächtige Kommunikationsmittel.

Vertreten von: Kahn und Saraswat [14], Raeder [21], Shu [23].

Diese These beruht auf der Annahme, daß die mehrdimensionale Informationskodierung bei Bildern (Position, Form, Farbe, Helligkeit usw.) deren Ausdrucksstärke gegenüber Text erhöht, sodaß der Bedeutungsgehalt pro Bildeinheit höher ist, als jener eines verbalen Ausdrucks. Was dabei unbeachtet bleibt, ist die entscheidende Rolle von Konventionen. Damit die vom Sender aufbereitete (verschlüsselte) Information vom Empfänger verstanden (entschlüsselt) werden kann, müssen Sender und Empfänger eine Übereinkunft über das verwendete Zeichensystem, die

Kodiervorschriften und den Begriffsraum treffen. Diese Bereiche sind bei verbaler Kommunikation weitgehend normiert, bei visueller Kommunikation hingegen nicht.

Es kann nicht bestritten werden, daß eine große Klasse graphischer Darstellungen effizient Informationen vermittelt, die textuell nur schlecht repräsentierbar sind. Graphische Repräsentationen von Meßdaten sind beispielsweise im allgemeinen bedeutend klarer und dichter, als die Darstellung in Form von Zahlenreihen. Die graphische Darstellung von Zahlenreihen ist jedoch ein Beispiel für Datenvisualisierung und nicht für VP. Weiters ist zu bedenken, daß in der Programmierung häufig mathematische Fragestellungen interessieren. In diesem Fall sind Bilder (etwa in Form von Diagrammen) textuellen Darstellungen bezüglich Präzision, Dichte und Handhabbarkeit klar unterlegen. Die bildliche Darstellung wird zwar benötigt, um den mathematischen Hintergrund zu illustrieren, kann aber meist nur einen Ausschnitt des Wertebereichs zeigen, der durch einen Algorithmus oder eine Formel definiert wird. Bilder mathematischer Sachverhalte sind demzufolge nur selten vollständig und bedeuten Informationsverlust, was der These vom “mächtigen Kommunikationsmittel” widerspricht.

Trotz der genannten Bedenken gibt es ernstzunehmende Hinweise, daß in bestimmten Anwendungsbereichen visuelle Programmiersprachen von entscheidender Bedeutung für die Verständigung von Entwicklern und Auftraggebern sein können. Ein hohes Kommunikationspotential liegt vor allem dann vor, wenn Ähnlichkeiten zwischen Programmiersprache und Darstellungen existieren, mit denen die Auftraggeber vertraut sind. Baroth und Hartsough [2] berichten, daß bei der Entwicklung von Meßinstrumenten mit LabVIEW, die Trennung der Phasen Anforderungsdefinition und Implementierung vollständig entfällt. Durch die ständige Kommunikation anhand des graphischen LabVIEW-Codes, der elektronischen Schaltplänen ähnelt und im Beisein des Auftraggebers interaktiv am Computer entwickelt wird, fließen Änderungen der Anforderungen sofort in das Produkt ein. Wenn alle Anforderungen erfüllt sind, ist auch das System fertig. Bei diesem Entwicklungsprozeß überblickt der Kunde zwar nicht alle Details der visuellen Syntax, er erkennt und versteht aber die Grobstruktur des Systems soweit, daß er an entscheidenden Stellen Vorschläge machen kann. Baroth und Hartsough meinen, daß eine solche Interaktion zwischen Entwickler und Auftraggeber auf Basis einer textuellen Sprache kaum vorstellbar ist, weil diese von den Auftraggebern nicht verstanden würde.

Fazit: Bilder können nur eine Ergänzung zu verbaler Verständigung sein. Nicht alles ist bildlich darstellbar, was verbal beschreibbar ist. Im formalen Bereich ist Graphik alleine kaum jemals ausdrucksstärker als Text.

3.2 These: Bilder sind leicht verständlich.

Vertreten von: Myers [17], Shu [23].

Zweifellos tragen gute Graphiken viel zur besseren Veranschaulichung von Informationen bei. Tufte [28] zeigt anhand vieler Beispiele, wie sorgfältig erstellte Diagramme, Tabellen, Karten, Pläne usw. komplexe Daten und Zusammenhänge so geschickt präsentieren können, daß der Betrachter mühelos die wesentlichen Informationen entnehmen kann. In der Programmierung können Bilder ebenfalls viel zum leichteren Verstehen beitragen, vor allem wenn es darum geht, durch geeignete Visualisierungen die Wirkungsweise von Programmen darzustellen. Durch die hohe Rechenleistung moderner Computer sind nicht nur statische Graphiken, sondern auch animierte Sequenzen möglich, durch die man einen unmittelbaren Eindruck von den Geschehnissen im "Inneren" eines Programms erhalten kann. Den großen Wert von Visualisierungen in der Programmierung unterstreicht auch Knuth [15], wenn er im ersten Band der Reihe "The Art of Computer Programming" feststellt: "An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it".

Die verständliche Darstellung von Programmcode in graphischer Form ist leider nur schwer realisierbar. So sind etwa die Ergebnisse von Laboruntersuchungen zur Lesbarkeit von Flußdiagrammen nicht ermutigend (vgl. Green und Petre [11]) und auch moderne Ansätze in Form von Datenflußdiagrammen und objektorientierten, visuellen Sprachen können nicht immer überzeugen, wie Abbildung 2 unterstreicht. In Situationen, wo eine bekannte Metapher für die Programmierung verwendet wird, besteht zudem auch die Gefahr, daß durch die visuell ansprechende und vertraute Form von Diagrammen die dahinterliegende Logik nur scheinbar verstanden wird. Petre [19] nennt dies den "Cobol Effekt": weil das Vokabular geläufig ist, glaubt man das Programm zu begreifen, obwohl man seine Funktionsweise in Wirklichkeit mißversteht.

In der objektorientierten Programmierung ist eine verständliche graphischen Darstellung von Programmcode besonders schwer zu erzielen. Man könnte meinen, daß die realitätsnahe Modellierung der objektorientierten Programmierung die anschauliche Darstellung von

Programmcode leicht machen müßte: in vielen objektorientierten VP-Umgebungen wird der Ansatz verfolgt, Objekte durch Piktogramme oder Benutzungsschnittstellen-Elemente darzustellen und Nachrichten durch beschriftete Pfeile. Diese Art der graphischen Darstellung hat allerdings den Nachteil, daß bereits kleine Diagramme völlig unübersichtlich werden. Abbildung 2 zeigt eine einfache, visuelle PARTS-Applikation mit allen Objekten und Nachrichten. Das Programm ist völlig undurchschaubar und kann nur mit größter Mühe besser strukturiert werden.

Das Linienwirrwarr ist leider inhärent mit dem Programmierparadigma verbunden. Wie die meisten objektorientierten Applikationen sind auch PARTS-Applikationen ereignisgesteuert. Immer wenn ein Ereignis eintritt, wird mit einer Kette von Nachrichten reagiert, die im allgemeinen viele Objekte berührt. Jeder Linienzug, der eine Nachrichtenkette visualisiert, muß demnach durch die Piktogramme aller beteiligten Objekte geführt werden. Weil auch sehr einfache Applikationen auf eine Vielzahl von Ereignissen reagieren, führen meist dutzende Linienzüge kreuz und quer durch die Diagramme. Auf diese Weise entsteht der verheerende Spaghetti-Effekt. Bei textuellen Notationen ist das nicht der Fall, weil dort Objekte über Namen angesprochen werden und daher keine visuelle Verbindung zwischen der Deklaration und der Benutzung eines Objekts besteht. Auch Datenflußdiagramme leiden nicht so stark unter dem Spaghetti-Effekt. Bei diesen graphischen Notationen für erweiterte funktionale Programmiersprachen (vgl. Hils [12]) werden Funktionen anstelle von Objekten miteinander verbunden. Anders als Objekte haben Funktionen kein Zustandsgedächtnis und deshalb kann eine Funktion mehrmals im Diagramm plaziert werden, was einem mehrmaligen Funktionsaufruf entspricht.

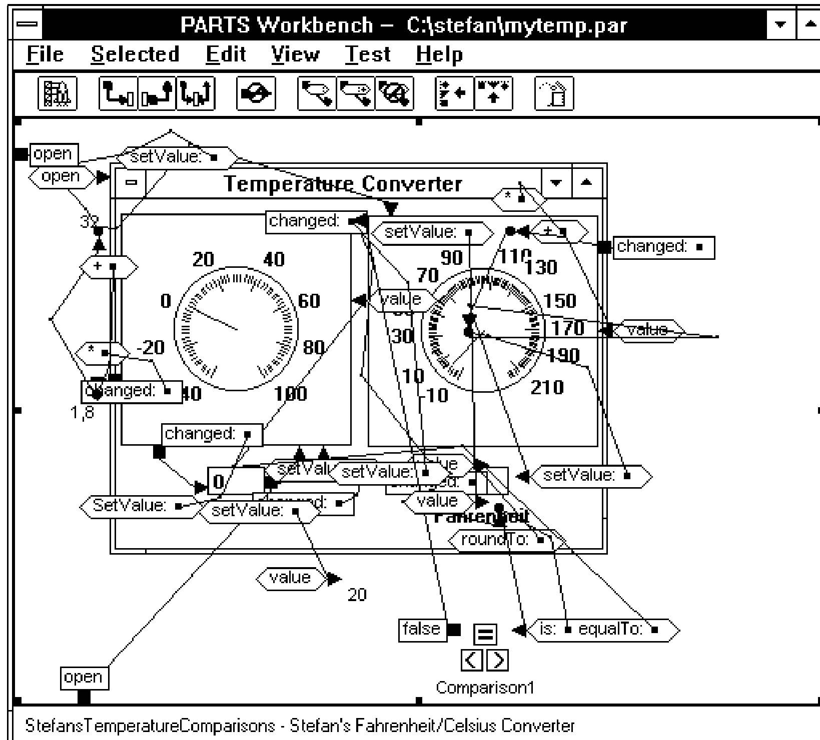


Abbildung 2: Ein PARTS-Programm zur Umrechnung von Celsius- und Fahrenheit-Werten.

Fazit: Die Darstellung von Programmcode in graphischer Form ist oftmals eher verständnishemmend als verständnisfördernd. Dies gilt insbesondere für Diagramme, wo Beziehungen zwischen einzelnen Elementen durch Linien dargestellt werden. In solchen Diagrammen führen komplexere Strukturen unausweichlich zu unentwirrbaren Liniengeflechten.

3.3 These: Bilder kann man sich leicht merken.

Vertreten von: Glinert [8], Shu [23], Staufer [25].

Das visuelle Gedächtnis des Menschen besitzt zweifellos eine hohe Kapazität. Anderson [1] führt Experimente an, bei denen Versuchspersonen aus tausenden Bildern einen hohen Prozentsatz bei Wiedervorlage richtig erkannten. Bei der Aufnahme bildlicher Information bleiben jedoch visuelle Einzelheiten oder räumliche Beziehungen von Bildelementen kaum im Erinnerung. Statt der bildlichen Darstellung prägt sich eine abstrakte Repräsentation der Bedeutung des Bildes ein, deren Codierung durchaus alle visuellen Attribute verloren haben kann. Nicht interpretierbare Bilder kann man sich deshalb ebenso schlecht merken, wie nicht interpretierbare verbale Information.

In der Softwareentwicklung sind die Anforderungen an das Gedächtnis weit gestreut. Zu den Dingen, die ein Programmierer im Kopf behalten muß, gehören unter anderem der Aufbau und die Funktionalität des Programms, typische Programm- und Datenstrukturen, Entwurfs- und Codierstrategien, Syntax und Semantik von Programmiersprachen, Bedienung der Werkzeuge, Datei- und Verzeichnisnamen. Nur weniges aus diesen Bereichen läßt sich durch einprägsame Bilder darstellen.

Zu den erfolgversprechendsten Visualisierungen gehören Bilder der Systemarchitektur, Entwurfsskizzen und Datenstrukturdiagramme. Die Wirksamkeit von visuellem Programmcode ist hinsichtlich des Erinnerungsvermögens als gering einzuschätzen. Dies läßt sich aus dem zuvor genannten Zusammenhang von Bedeutungsgehalt und Gedächtnisleistung ableiten, der es äußerst zweifelhaft erscheinen läßt, daß bildliche Darstellungen von Programmen besser behalten werden, als textuelle Repräsentationen. Die Bedeutung eines Programms anhand einer bildlichen Repräsentation auf einen Blick zu erfassen und zu merken ist wohl unmöglich, da sich die Semantik aus vielen Details ergibt. Wie Anderson zeigt, werden gerade solche Details sehr schnell vergessen.

Fazit: Vom gutem Erinnerungsvermögen für Bilder aus dem Alltag darf nicht geschlossen werden, daß auch visuelle Programme gut behalten werden. Um sich etwas merken zu können, muß man dessen Bedeutung erfaßt haben, was insbesondere bei detailreichen Codestücken schwierig ist. In diesen Fällen behält man zwar eventuell die Grobstruktur im Gedächtnis, wichtige Informationen, wie Abbruchbedingungen für Rekursionen und Schleifen, Assertionen usw. gehen aber im graphischen Dickicht verloren. Eine verbale Repräsentation prägt sich in diesen Fällen vermutlich leichter ein.

3.4 These: Bilder erleichtern das Lernen.

Vertreten von: Glinert [8], Myers [17], Shu [23].

Bilder sprechen die spielerische Seite im Menschen an und scheinen deshalb für das Erlernen von Programmiersprachen ein hohes Motivationspotential aufzuweisen (vgl. Tanimoto und Runyan [27]). Viele visuelle Programmierumgebungen und -sprachen sind mit dem Ziel entstanden, Kindern und Anfängern den Einstieg in die Programmierung zu erleichtern.

Um die These der leichten Erlernbarkeit zu überprüfen, soll hier allerdings nicht Ad-hoc-Programmierung betrachtet werden, sondern der seriöse Programmierunterricht als Beurteilungsgrundlage dienen. In diesem Zusammenhang ist das Ausbildungsziel maßgebend, denn hohes Motivationspotential alleine darf kein Grund sein, eine visuelle Programmiersprache zu wählen. Von einer sinnvollen Einführung in die Programmierung erwartet man, daß sie die Grundlagen für den Umgang mit komplexen Problemstellungen legt und die dazu notwendigen Konzepte, Prinzipien und Techniken vermittelt. Die verwendete Programmiersprache soll diese Lehrinhalte unterstützen und möglichst einfach sein. Bilder können zwar helfen, gewisse Aspekte zu verdeutlichen und somit die Erlernbarkeit verbessern, bringen aber auch Nachteile mit sich. So zeigen Ablaufdiagramme zwar gut die strukturellen Eigenschaften von Algorithmen, es können bei undiszipliniertem Gebrauch aber unübersichtliche Strukturen entstehen. Gerade bei Programmieranfängern ist Disziplin jedoch eher selten anzutreffen. Die Verwendung graphischer Hilfsmittel, die einen großen Freiheitsgrad in der Programmgestaltung erlauben, kann daher in der Lernphase zur Fixierung eines schlechten Programmierstils führen.

Dijkstra [7] lehnt Visualisierungen strikt ab, weil er dadurch das Abstraktionsvermögen gefährdet sieht. Das Hochgefühl, ohne viel Nachzudenken durch Kombination weniger Piktogramme in Minutenschnelle ein simples Programm erstellen zu können, kann tatsächlich trügerisch sein. Entscheidend ist, ob das Gelernte ausreicht, auch schwierigere Probleme zu meistern. Beobachtungen des Autors als Leiter von Programmierprojekten, bei denen sowohl visuelle als auch verbale Sprachen zur Lösung der selben Aufgabe eingesetzt wurden, zeigten ab einer gewissen Problemkomplexität einen signifikanten Einbruch der Programmierleistung mit visuellen Sprachen. Dazu der Kommentar eines Praktikumsteilnehmers:

> *Die visuelle Programmiersprache Serius ist eine jener Sprachen, bei der man mit sehr geringem Aufwand relativ gute bzw. sichtbare Erfolge erzielen kann, allerdings ist jede weitere Verbesserung umso schwieriger. Diesem schnellen Erfolgserlebnis war auch ich aufgelaufen, wodurch meine erste Version des Taschenrechners nur schwer erweiterbar war, da ich auch nicht annahm, daß Erweiterungen geplant waren. Den größten Rückschlag erlitt ich, als ich eine Funktion von Smalltalk nach Serius übertragen wollte. Nach stundenlanger Arbeit mußte ich einsehen, daß dies fast unmöglich war, da ich für eine einzige Smalltalk-Zeile (Zuweisung mit indizierten Feldern) ca. zehn Serius-Funktionen verwenden mußte, was durch die geringe Bildschirmgröße noch zusätzlich verschlimmert wurde.*

Fazit: Es ist richtig, daß visuelle Programmiersprachen durch Piktogramme und interaktive Manipulation motivierend auf Anfänger wirken, weil sie syntaktische Aspekte zurückdrängen und

einen spielerischen Umgang mit den Sprachkonstrukten erlauben. Der anfängliche Enthusiasmus für visuelle Programmiersprachen schlägt jedoch rasch in Frustration um, wenn geeignete Sprachkonzepte zur Bewältigung komplexer Problemstellungen fehlen oder umständliche Interaktionsmechanismen das schnelle Editieren größerer Programme verhindern. Solche Mängel erzwingen den Umstieg auf eine andere, meist textuelle Sprache, wodurch ein guter Teil des Lernaufwandes umsonst war.

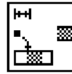
3.5 These: Bilder stoßen auf keine Sprachbarrieren.

Vertreten von: Kahn und Saraswat [14], Shu [23], Suleiman und Citrin [26].

Shu versucht diese These mit dem Beispiel von Verkehrszeichen zu untermauern, die überall auf der Welt verstanden werden. Es stimmt, daß unter den autofahrenden Nationen sich sowohl Verkehrssituationen als auch Verkehrszeichen ähneln und letztere deshalb über Sprachgrenzen hinweg verständlich sind; unklar ist hingegen der Zusammenhang mit den Kommunikationsbedürfnissen in der Programmierung. Die Analogie zwischen Verkehrszeichen und Programmkonstrukten gilt bestenfalls für die Schlüsselworte einer Programmiersprache, die ein begrenztes Vokabular mit einer genau definierten Bedeutung umfassen. Internationale Anstrengungen vorausgesetzt könnte man eindeutige Piktogramme für die gängigsten Schlüsselworte finden, sodaß diese unabhängig von den nationalen Sprachen ein genormtes Erscheinungsbild hätten. Wozu aber solche Bemühungen? Die Herausforderung bei der Entwicklung von Programmen liegt ja nicht im Verstehen der Sprachkonstrukte, sondern in der Bewältigung der Komplexität der Programmbausteine.

Ungleich schwieriger als das Verstehen von Schlüsselworten, ist die aussagekräftige Benennung von Programmeinheiten wie Variablen, Prozeduren, und Module. Professionelle Programmierer legen hohen Wert darauf, sprechende Namen zu vergeben, weil sie wissen, daß davon zu einem guten Teil das Programmverständnis abhängt. Dies hat selbstverständlich in einer lebenden Sprache zu geschehen, da nur auf diese Weise ein ausreichender Bedeutungsgehalt gewährleistet ist. Es ist völlig ausgeschlossen, daß in einem offenen Begriffsraum, wie ihn die Programmierung von Softwaresystemen umfaßt, Bilder jemals den gleichen Standardisierungsgrad und die gleiche Aussagekraft wie die Worte einer natürlichen Sprache erreichen. Tausende, täglich neu geschaffene Bezeichnungen für Programmfunktionen und Softwarekomponenten können niemals in einem

internationalen Lexikon der Bildersprache verzeichnet werden. Zur Illustration möge sich der Leser fragen, was dieses (wohldurchdachte) LabVIEW-Symbol bedeutet:



Dieses Symbol repräsentiert die Funktion *Substring(string,offset,length)*. Jeder Programmierer wird nun wissen, welche Bedeutung dem Piktogramm zugrunde liegt.

Fazit: Den gesamten Begriffsraum der Programmierung in Bildern zu fassen ist unmöglich. Die natürliche Sprache mit ihren differenzierten Ausdrucksmöglichkeiten ist in der Programmierung unverzichtbar, um Dinge zu benennen und Lösungsideen zu vermitteln. Ein Bilder-Esparanto würde unüberwindliche Kommunikationsbarrieren errichten, die jede Verständigung zum Erliegen brächten.

Ausgehend von den zuvor angeführten Überlegungen, versuchen die nächsten Abschnitte das Potential und die Grenzen visueller Darstellungen in der Programmierung abzustecken.

4. Potential visueller Darstellungen in der Programmierung

Die Stärken von Graphiken in der Softwareentwicklung liegen vor allem in jenen Bereichen, wo die Vermittlung von Ideen, die Darstellung komplexer Zusammenhänge und die Gestaltung von Benutzungsschnittstellen im Mittelpunkt stehen. Dabei kommen unter anderem folgende positive Eigenschaften bildlicher Darstellungen zur Geltung:

- Effektive Verarbeitung visueller Information im Gehirn
- Anschaulicher Realitätsbezug
- Erhöhter Aufmerksamkeitswert
- Großes Motivations- und Lernpotential
- Abschwächung syntaktischer Strukturen
- Betonung semantischer Zusammenhänge
- Fehlervermeidung durch direkte Manipulation

Die nächsten Abschnitte skizzieren, wie diese Vorteile in der Softwareentwicklung genutzt werden können. VP im engeren Sinn, d.h. die Erstellung von Programmcode mit graphischen Notationen,

spielt dabei kaum eine Rolle. Die angeführten Potentiale visueller Darstellungen können ebenso bei verbaler Programmierung genutzt werden.

4.1 Überblicksbilder und Entwurfskizzen

Für die Spezifikation, den Entwurf und die Dokumentation eines Softwaresystems sind anschauliche Graphiken ein bedeutender Faktor zur Verbesserung des Problemverständnisses, zur Förderung des Ideenaustausches sowie zur Darstellung der Struktur und Funktionalität des Systems.

Dabei ist entscheidend, daß sich Zeichnungen und Diagramme auf die essentiellen Aspekte des betreffenden Sachverhalts beschränken, um den Betrachter nicht mit irrelevanten Details zu belasten. Dies geschieht am besten mit informellen Mitteln, wie anschaulichen Piktogrammen, eingängigen Symbolen und verbalen Erläuterungen. Formale Notationen sind dazu nicht notwendig, sondern stören eher. Streng genormte Graphiken überlagern semantische Informationen durch syntaktische Konstrukte und behindern dadurch den Konstruktions- und Wahrnehmungsprozeß. In der Vergangenheit wurde zwar immer wieder versucht, strikt definierte und allgemeingültige Entwurfsnotationen zu finden, die Erfahrung aber zeigt, daß dies auch für abgegrenzte Bereiche problematisch ist. Die Kreativität wird dadurch oft in ein zu enges Korsett gezwängt. Darauf verweist auch Denert [6] in einer kritischen Auseinandersetzung mit CASE-Werkzeugen: "Man schreibt nur noch das auf, wofür das Werkzeug eine passende Schublade hat. Die unausgegorenen, aber möglicherweise besonders guten Ideen werden vielleicht gar nicht mehr festgehalten, denn man hat ja eigentlich nur die Aufgabe, die Datenbasis des Werkzeugs zu füllen. Und das heißt: man kann ungeheuer beschäftigt sein, ohne wirklich voranzukommen. Man sieht sich satt an den vielen gleichartigen Bildern".

4.2 Softwarevisualisierung

Graphische Repräsentationen der dynamischen Programmstruktur sind beim Test, bei der Fehlersuche und bei der Optimierung von Laufzeit- und Speicherbedarf von entscheidendem Vorteil, da sie Zusammenhänge erschließen, die bei einer Inspektion der statischen Struktur verborgen bleiben.

Besonders einprägsam sind Animationen, die Änderungen des Programmzustandes in Form von bewegten Bildern visualisieren. Der damit mögliche Einblick in die Programmausführung ist

speziell in Systemen mit parallelen Prozessen von größter Bedeutung, da dort oftmals zeitliche Abhängigkeiten zwischen den einzelnen Ausführungseinheiten bestehen und das Zusammenspiel der Programmkomponenten ohne direkte Beobachtung nur schwer nachvollziehbar ist.

Ein anderes essentielles Problem in der Programmierung ist die Laufzeitabschätzung. Einsichten in die Laufzeitkomplexität eines Berechnungsverfahrens können sogenannte Algorithmenrennen vermitteln. Dabei werden verschiedene Algorithmen für das selbe Problem und die selben Eingangsdaten gleichzeitig gestartet und simultan animiert. Durch Beobachtung des “Rennens” kann man meist auf Anhieb erkennen, welches Verfahren effizienter ist. Gute Visualisierungen können auch das Verständnis für objektorientierte Systeme bedeutend erhöhen, indem sie den Lebenslauf von Objekten sichtbar machen. Durch Interaktion mit der Applikation und gleichzeitiger Überwachung der Visualisierung kann der Programmierer feststellen, welche Aktionen neue Objekte erzeugen, welche Objekte zu welchen Zeitpunkten intensiv miteinander kommunizieren, welche Auswirkungen der Polymorphismus auf den Kontrollfluß hat usw. Dabei kommt es nicht auf eine möglichst konkrete Repräsentation der involvierten Objekte an, sondern auf eine Darstellung, welche die Analyse der interessierenden Vorgänge bestmöglich unterstützt und durchaus abstrakt sein kann.

4.3 Graphische Benutzungsschnittstellen

Es ist unumstritten, daß graphische Benutzungsschnittstellen mit ergonomischer Gestaltung die Akzeptanz und Wirksamkeit eines Softwaresystems gegenüber zeichenorientierten Schnittstellen in der Regel um ein Vielfaches erhöhen.

Die Interaktion zwischen Benutzer und System erfolgt bei graphischen Schnittstellen über einprägsame Elemente wie Befehlsknöpfe, Ein/Ausschalter, Schieberegler und visuellen Anzeigeneinheiten, die der Anwender zum Teil von technischen Geräten des Alltags kennt. Durch die unmittelbare optische Wirkung und die Möglichkeit der direkten Manipulation verfügen solche Schnittstellen über ein hohes Motivationspotential, wodurch die Bedienung weniger anstrengend empfunden wird als bei Systemen, die auf textueller Kommandoeingabe beruhen. Eine Reihe weiterer Vorteile, wie leichte Erlernbarkeit, intuitive Handhabung, Reduktion von Fehlerquellen und Anpaßbarkeit an persönliche Bedürfnisse, begründen die überragende Stellung graphischer Benutzungsschnittstellen als Mechanismus für die Interaktion zwischen Anwender und Applikation.

Verglichen mit dem Aufwand, der für die Realisierung von Benutzungsschnittstellen in einer verbalen Programmiersprache nötig ist, sind die Entwicklungszeiten mit Spezialwerkzeugen aus dem Bereich der VP drastisch reduzierbar.

5. Grenzen visueller Darstellungen in der Programmierung

Die Schwächen von Graphiken in der Softwareentwicklung liegen vor allem in jenen Bereichen, wo die eindeutige Darstellung von Sachverhalten, die rasche und bequeme Änderung komplexer Beschreibungen, die werkzeugunterstützte Bearbeitung von Dokumenten sowie die Formalisierung von Methoden und Notationen im Mittelpunkt steht. Dabei wirken sich unter anderem folgende Eigenschaften bildlicher Darstellungen nachteilig aus:

- Keine Standards für Symbole, Notationen und Werkzeuge
- Geringe Darstellungsdichte
- Geringer Strukturierungsgrad
- Hoher Formulierungsaufwand
- Hohe Resistenz gegenüber Modifikationen
- Hohe ästhetische Anforderungen
- Schwer zu formalisieren

Diese Mängel betreffen leider weite Bereiche der Implementierungsphase. Auf der Suche nach durchschaubaren und effektiven Techniken zur Implementierung von Qualitätssoftware ist daher VP wenig erfolgversprechend. Die nächsten Abschnitte skizzieren, wie die erwähnten Schwachstellen in der Softwareentwicklung zum Tragen kommen.

5.1 Fehlende Standards

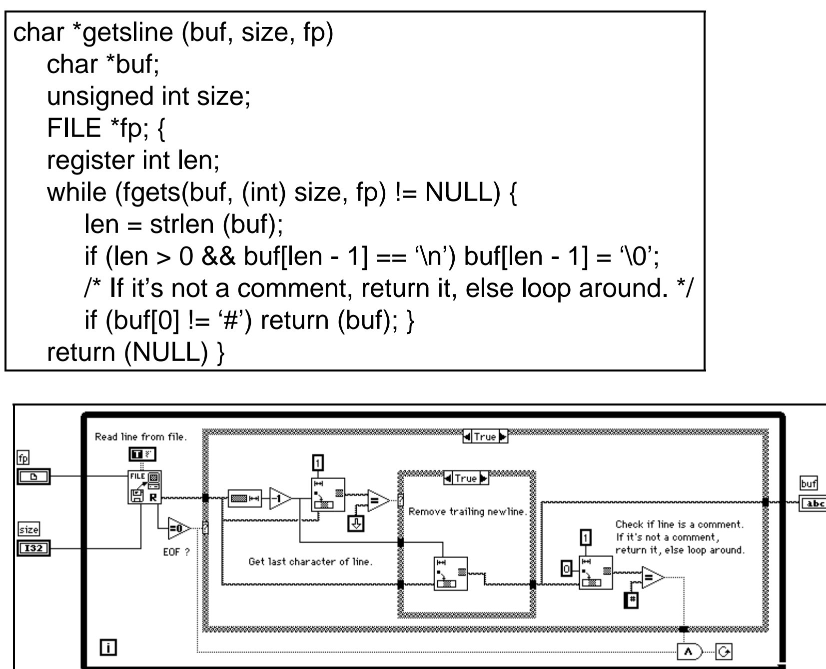
Für textuelle Notationen gibt es eine Reihe von Normen und ungeschriebenen Übereinkommen, ohne deren Existenz die wirtschaftliche Herstellung von Software undenkbar wäre. Gängige Standards umfassen Bereiche wie Zeichenkodierungen, Dokumentstrukturen und Metasprachen.

Für allgemeine visuelle Notationen im Bereich der Softwaretechnik gibt es kaum verbindliche Regeln, sieht man von Normen für bestimmte Darstellungen ab, zu denen etwa Flußdiagramme oder SDL-Charts gehören. Mangels Normen und De-facto-Standards existieren auch kaum Werkzeuge

mit einheitlichen Schnittstellen, die visuelle Softwarebeschreibungen und Programme austauschen, interpretieren und verarbeiten können. Jeder Hersteller von VP-Produkten entwickelt ein eigenes graphisches Vokabular und proprietäre Datenstrukturen, auf die alle Werkzeuge der Entwicklungsumgebung abgestimmt sein müssen. Die Werkzeuge eines Herstellers bilden somit ein geschlossenes System, das mit keiner anderen Umgebung kommunizieren kann. Eine Übereinkunft zu Darstellungsstandards, Austauschformaten und visuellen Programmiersprachen liegt in weiter Ferne. Der Käufer eines VP-Systems begibt sich damit in eine starke Abhängigkeit vom Hersteller mit allen damit verbundenen Nachteilen.

5.2 Unökonomische Bildschirmnutzung

Ein notorischer Schwachpunkt graphischer Notationen ist die unökonomische Nutzung der zur Verfügung stehenden Darstellungsfläche. Nickerson [18] hat sich um Metriken für die Darstellungsdichte von textuellen und graphischen Notationen bemüht und kommt zum Schluß, daß wegen des begrenzten Auflösungsvermögens von Ausgabeeinheiten und Auge rein textuelle Notationen immer dichter sind, als gemischt graphisch/textuelle Repräsentationen.



```
char *getsline (buf, size, fp)
    char *buf;
    unsigned int size;
    FILE *fp; {
    register int len;
    while (fgets(buf, (int) size, fp) != NULL) {
        len = strlen (buf);
        if (len > 0 && buf[len - 1] == '\n') buf[len - 1] = '\0';
        /* If it's not a comment, return it, else loop around. */
        if (buf[0] != '#') return (buf); }
    return (NULL) }
```


Abbildung 3. Annähernd flächengleiche Darstellung von textuellem und graphischem Programmcode der gleichen Funktion.

Abbildung 3 zeigt die annähernd flächengleiche Darstellung der textuellen und der graphischen Version einer einfachen Funktion. Während die textuelle Repräsentation problemlos lesbar ist, kann die graphische Darstellung bei einer Standardbildschirmauflösung von 72 dpi nicht entziffert werden. Die Graphik ist aber nicht nur viel schwieriger zu lesen als der Text - es ist auch unmöglich, eine derart verkleinerte Graphik zu modifizieren. Die Darstellung des graphischen Programmcodes in lesbarer Originalgröße nimmt 772x254 (ca. 600.000) Pixel ein und braucht somit ca. 18 mal mehr Platz als der textuelle Programmcode mit 264x126 (ca. 33.000) Pixel. Die Pixelwerte basieren auf den Abmessungen der schwarzen Rahmen und umfassen somit auch Leerraum.

Wegen der geringen Dichte graphischer Notationen ist es äußerst schwierig, einen Gesamteindruck von einem nicht-trivialen Programm zu bekommen. Selbst wenn ein visuelles Programm auf Papier ausgedruckt vorliegt, verbessert sich die Situation nicht. Im Gegenteil: Programmteile, die nicht auf ein einzelnes Blatt Papier passen, müssen wie ein Puzzle zusammengeklebt werden. Diese mühevolle und fehleranfällige Prozedur steht dem Bedürfnis nach wirtschaftlicher Softwareentwicklung diametral entgegen.

5.3 Schlechte Skalierbarkeit

Die Skalierbarkeit einer Sprache gibt an, inwieweit sie zur Beschreibung einfacher und komplexer Sachverhalte gleichermaßen gut geeignet ist. Eine gute Skalierbarkeit geht Hand in Hand mit einem Formulierungsaufwand, der relativ zur Problemgröße in etwa gleich bleibt. Der Formulierungsaufwand ist ein Maß dafür, wieviele Details die Beschreibung einer Problemlösung umfaßt.

Im Vergleich zu Sprachen auf textueller Basis ist die Skalierbarkeit visueller Programmiersprachen meist gering. Der Grund für den hohen Formulierungsaufwand liegt in den geringen optischen Strukturierungsmöglichkeiten bei gleichzeitig hohen ästhetischen Anforderungen und wird durch die hohe Resistenz gegenüber Modifikationen zusätzlich verschärft. Durch die schlechte Skalierbarkeit verlangt die Lösung komplexer Probleme unverhältnismäßig große Anstrengungen. Poswig [20] weist in diesem Zusammenhang darauf hin, daß bei visuellen Programmiersprachen der Formulierungsaufwand nicht an der Mächtigkeit der Sprache alleine gemessen werden darf, sondern auch die Interaktion mit dem Graphikeditor einbezogen werden muß, der bei der Programm-

konstruktion zum Einsatz kommt. Der Interaktionsaufwand für graphische Editoren ist meist erheblich höher als für Texteditoren.

5.4 Schwierige formale Handhabbarkeit

Ein wesentliche Schwäche visueller Programmiersprachen ist das Fehlen von ausgereiften Formalismen zur Definition von Grammatiken, semantische Bedingungen und Transformationsregeln. Visuelle Sprachen weisen durch ihren mehrdimensionalen Charakter eine Vielzahl von Eigenschaften auf, die es schwierig machen, eine Metasprache zu finden, die alle Aspekte gleichermaßen gut abdeckt und trotzdem verständlich ist.

Während in verbalen Sprachen die lexikalischen Grundeinheiten aus einfachen Symbolen (Zeichen oder Worte) bestehen und die syntaktische Beziehung zwischen den Symbolen ausschließlich durch die Hintereinanderaufschreibung festgelegt wird, ist die Situation bei visuellen Sprachen vielschichtiger: hier können verschiedenen Klassen von Beziehungen auftreten, die kaum mit einem einzigen Formalismus abgedeckt werden können. Dazu gehören etwa geometrische Relationen (z.B. Größenverhältnisse und Positionen), topologischen Relationen (z.B. Vernetzungen, Berührungen und Schachtelungen) und dynamische Aspekte (z.B. Bewegung, Farbänderungen und Blinken). Insbesondere dynamische Aspekte, die in VP-Systemen eine große Rolle spielen, sind mit statischen Beschreibungen nur schwer erfaßbar. Zudem sind Entwickler von Parsing-Algorithmen für visuelle Sprachen mit graphentheoretischen Problemen konfrontiert, die nur oft nur mit Verfahren exponentieller Laufzeitkomplexität lösbar sind und deshalb für praktische Zwecke nicht in Frage kommen. Mit Grammatiken und Compilern zu visuellen Sprachen beschäftigen sich u.a. Chang et al. [4], Costagliola et al. [5], Golin und Reiss [10], Rekers und Schürr [13], Meyer [16] und Wittenburg [29].

6. Zusammenfassung

Die Stärken der VP liegen in speziellen Anwendungsgebieten, wo überschaubare und abgegrenzte Problemstellungen durch visuelle Metapher gut erfaßbar sind. Beispiele dafür sind die Programmierung virtueller Instrumente und Bereiche der Automatisierungstechnik. In Spezialgebieten kann VP auch Anwender in die Lage versetzen, kleine und überschaubare Programme zu erstellen, die sonst Softwareingenieure schreiben müßten. Ebenfalls von hohem Wert

sind graphische Darstellungen softwaretechnischer Sachverhalte in Form von Entwurfsskizzen und Visualisierungen, wenn auf Details zugunsten der Verständlichkeit verzichtet wird. Ob der zugrundeliegende Programmcode verbal oder visuell erstellt wurde, ist in diesem Zusammenhang von untergeordneter Bedeutung.

Zur Erstellung komplexer Programmstrukturen ist VP wegen der schlechten Skalierbarkeit nur bedingt geeignet. Motivierende Interaktionsmechanismen (z.B. direkte Manipulation und ansprechende Symbolbilder) können zur Belastung werden, wenn große Komponenten zu erstellen sind. Petre [19] zitiert einen ernüchterten Programmierer mit folgenden Worten: “I quite often spend an hour or two just moving boxes and wires around, with no change in functionality, to make it that much more comprehensible when I come back to it.” Auf der Suche nach Ansätzen zur professionellen und rationellen Entwicklung allgemeiner Softwaresysteme ist VP deshalb als Sackgasse zu beurteilen.

Literatur

[1] ANDERSON, J.R., Kognitive Psychologie: Eine Einführung, Spektrum der Wissenschaft Verlagsgesellschaft 1989.

[2] BAROTH, E., HARTSOUGH, C., Visual Programming in the Real World, in: Margaret M. Burnett, Adele Goldberg, Ted G. Lewis (ed.): Visual Object-Oriented Programming, Concepts and Environments, Manning, Prentice Hall 1995, 21-42.

[3] CHANG, S.K. (ed.), Principles of Visual Programming Systems, Prentice Hall 1990.

[4] S.K. CHANG, S.K., TAUBER, M.J., B. YU, B., J.S. YU, J.S., A visual language compiler, IEEE Transactions on Software Engineering, Vol. 15, No. 5, May 1989, 506-525.

[5] COSTAGLIOLA, G., TORTORA, G., OREFICE, S., DE LUCIA, A. Automatic Generation of Visual Programming Environments, Computer, Vol. 28, No. 3, March 1995, 56-66.

[6] DENERT, E., Dokumentorientierte Software-Entwicklung, Informatik-Spektrum, Vol. 16, 1993, 159-164.

[7] DIJKSTRA, E.W., On the Cruelty of Really Teaching Computing Science, Communications of the ACM, Vol. 32, No. 12, Dec. 1989, 1398-1404.

[8] GLINERT, E.P., Nontextual Programming Environments, in: Shi-Kuo Chang (ed): Principles of Visual Programming Systems, Prentice Hall 1990, 144-230.

[9]GLINERT, E.P., TANIMOTO, S.L., Pict: An Interactive Graphical Programming Environment, Computer, November 1984, 7-25.

[10] GOLIN, E.J., REISS, S.P., The Specification of Visual Language Syntax, Journal of Visual Languages and Computing Vol. 1, No. 2, 141-157.

[11] GREEN, T.R.G., PETRE, M., When Visual Programs are Harder to Read than Textual Programs, Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference on Cognitive Ergonomics), G. C. van der Veer, M. J. Tauber, S. Bagnarola, M. Antavolits (ed.), Rome 1992.

[12] HILS, D.D., Visual Languages and Computing Survey: Data Flow Visual Programming Languages, Journal of Visual Languages and Computing, Vol. 2, No. 1, Mar. 1992, 69-101.

[13] REKERS, J., SCHÜRR, A., A Graph Grammar Approach to Graphical Parsing, in: Proceedings 1995 IEEE Symposium on Visual Languages, Sep. 5-9, 1995, Darmstadt, Germany, IEEE Computer Society Press, 195-202.

[14] KAHN, K.M., SARASWAT, V.A., Complete Visualizations of Concurrent Programs and their Executions, in: Proceedings 1990 IEEE Workshop on Visual Languages, Oct. 4-6, 1990, Skokie, Illinois, USA, IEEE Computer Society Press, 7-15.

[15] KNUTH, D.E., Fundamental Algorithms - The Art of Computer Programming, Addison-Wesley 1973.

[16] MEYER, B., Pictures Depicting Pictures: On the Specification of Visual Languages by Visual Grammars, in: Proceedings 1992 IEEE Workshop on Visual Languages, Sept. 15-18, 1992, Seattle, Washington, USA, IEEE Computer Society Press, 41-47.

[17] MYERS, B.A., Program Visualization, in: John J. Marciniak (ed.): Encyclopedia of Software Engineering, John Wiley & Sons, 1994, 877-892.

[18] NICKERSON, J.V., Visual Programming, Ph.D. Dissertation, New York University, 1994.

[19] PETRE, M., Why Looking Isn't always Seeing: Readership Skills and Graphical Programming, Communications of the ACM, Vol. 38, No. 6, June 1995, 33-44.

[20] POSWIG, J., Visuelle Programmiersprachen - Die Realisierung und konzeptionelle Weiterentwicklung eines Prototypen, Dissertation, Universität Dortmund, 1994.

[21] READER, G., A survey of current graphical programming techniques, Computer, Vol. 18, No. 8, August 1985, 11-25.

[22] SCHIFFER, S., FRÖHLICH, J.H., Visual Programming and Software Engineering with Vista, in: Margaret M. Burnett, Adele Goldberg, Ted G. Lewis (ed.): Visual Object-Oriented Programming, - Concepts and Environments, Manning Publications and Prentice Hall, 1995, 199-227.

[23] SHU, N.C., Visual Programming, Van Nostrand Reinhold 1988, 113-118.

[24] SNELL, M., Analysts predict $3.79 billion market for visual development tools by 1999, Computer, Vol. 28, No. 3, March 1995, 8-9.

[25] STAUFER, M., Piktogramme für Computer, de Gruyter 1987.

[26] SULEIMAN, K.A., CITRIN, W.V., An International Visual Language, in: Proceedings 1992 IEEE Workshop on Visual Languages, Sept. 15-18, 1992, Seattle, Washington, USA, IEEE Computer Society Press, 141-147.

[27] TANIMOTO, S.L., RUNYAN M.S., PLAY: An Iconic Programming System for Children, in: Shi-Kuo Chang, Tadao Ichikawa, Panos A. Ligomenides (ed.): Visual Languages, Plenum Press 1986, 191-205.

[28] TUFTE, E.R., Envisioning Information, Graphics Press 1990.

[29] WITTENBURG, K., Early-style Parsing for Relational Grammars, in: Proceedings 1992 IEEE Workshop on Visual Languages, Sept. 15-18, 1992, Seattle, Washington, USA, IEEE Computer Society Press, 192-199.