On Enhancing the Interface to the Source Code of Computer Programs

Ronald Baecker
Human Computing Resources Corporation
Toronto

Aaron Marcus
Aaron Marcus and Associates
Berkeley

Abstract
This paper addresses issues in the human factors of computer program documentation. We develop a framework for research on enhancing the interface to the source code of computer programs through designing and automating the production of effective typeset representations of the source text. Principles underlying the design research and examples of sample production are presented.

This work was supported by the U.S. Defense Advanced Research Projects Agency under DARPA Order 4469. We are grateful for valued assistance to the other members of the research team, Michael Arent, Design Director, Aaron Marcus and Associates, and Paul Breslin, John Jackson, Allen McIntosh, and Christopher Sturgess, Human Computing Resources Corporation, and to Trigraph Typesetters, Toronto.

Introduction

Although the art of enhancing in a task-independent manner the user interface to computer systems is continuously advancing, some of the strongest methods are intrinsically task-dependent. For example, displaying the user's input in common musical notation and playing it back through a loudspeaker in real time is a method of enhancing the interface to a musical score editor that does not apply directly to other interactive systems. Thus it is no surprise that there are numerous ways human factors can be applied to aid in the process of programming.

Historically, such efforts have taken a number of forms. The most widespread development has concerned the logical structure and expressive style of programs. Out of this concern have emerged many of the modern software development techniques, including top-down design and stepwise refinement [Wirth], structured programming [Dahl], modularity [Parnas], and software tools [Kernighan]. Another development has occurred in the organization of the editoral and production team that produces the writing, for example, the concepts of chief programmer teams [Baker] and structured walkthroughs [Yourdon]. A third development has been the introduction of various graphic notations for expressing algorithms such as Nassi-Shneiderman diagrams [Nassi], Warnier-Orr diagrams [Higgins], contour diagrams [Organick], and SADT diagrams [Ross]. A fourth and more recent development is supporting the writing and maintaining of good programs by providing integrated software development environments [Wasserman] and high-performance personal workstations specially for the task of program development [Teitelman, Deutsch, Gutz].

In our work we have taken a different approach [Marcus]. Since the advent of programming, the technologies of the video display terminal and the line printer have limited the presentation of computer program source code and comments to the use of a single type font, at a single point size, with single-width characters, and sometimes without even the use of upper and lower case. On the other hand, the technologies of high resolution bit map displays, laser printers, and computer-driven phototypesetters allow for the production of far richer representations, embodying multiple fonts, non-alphanumeric symbols, variable point sizes, proportional character widths, variable word spacing and line spacing, grey scale tints, rules, and arbitrary spatial location and orientation of elements on a page.

Thus we can take an entirely new approach to the presentation of source text: to make it more legible, more readable, more vivid, and more memorable. Although similar in purpose to work on pretty-printing [Oppen], it goes far beyond it in its methodology and its scope.

This paper will outline our research program and experimental methodology, and illustrate its application to the problem of enhancing the presentation of the source code of programs written in the C language.

We shall explain our approach to the design of C program text, and illustrate it with initial examples of its execution.

Research Program and Experimental Methodology

Our overall research program consists of six topics:

1. The first research topic deals with the appropriate use of typography to reveal formal syntactic, semantic, and pragmatic properties of programs and program elements.
2. A second concern is with the design and layout of program elements on the page using systems of grids, overlays, and windows.
3. A third area for research is the possibility of substituting a set of well-designed icons or symbols (pictograms or ideograms) for certain combinations of alphanumerics that occur repetitively in program code.
4. A fourth set of questions arise out of the possibilities that interactive computer graphics offer in the inclusion of movement, blinking, and other kinds of change into program documentation. More fundamentally, we must explore the relationship between static paper and dynamic screen representations of computer programs.
5. A fifth problem area is in the depiction of large directed graphs of great complexity, networks in which nodes are not single points but entire frames (combinations of signs) and in which links are explicitly stated or implied connections between nodes.
6. The final research topic concerns the ability of a program visualization to facilitate the integration of the various conceptual levels at which a program may be described.

Our work to date has centered on the first two of the above topics. We have adopted the following methodology:

1. We first developed a graphic design taxonomy for computer-based documents and publications. This was intended to be a checklist for approaches to the presentation of source code documentation.
2. We simultaneously developed a taxonomy of C constructs, a systematic enumeration and classification of aspects of the language. This was intended to be a companion checklist for insuring completeness in the representation of C source text.
3. Next, we collected and systematized typical mappings from C constructs to typographic constructs. These examples were abstracted from real C programs prepared by typical and for the most part experienced C programmers. We call such examples "folk designs".
4. Then, we developed a systematic approach to the design of mappings from C constructs to typographic constructs, an approach which forms the basis for detailed visual research into effective presentations of C source code.
5. Finally, we constructed a "first waffle" of a visual C compiler, a program which maps arbitrary C programs into effective typeset representations of the source code. We are now producing numerous examples using this automated tool.

These five steps are now described in more detail..

A Taxonomy of Typographic Constructs

In order to understand how source code might be displayed effectively in typeset form, it is necesary to analyze the graphic design possibilities of typography, the visual media of language expression. At a minimum this involves a systematic characterization of font selection, layout, and page sequencing. Many graphic design textbooks [Gerstner, Ruder] suggest a taxonomy of typographic form. Most large type display books show selections from the theoretical matrix of font, size, and line spacing possibilities. Graphic design manuals [Chaparos] display selections from the possibilities of page layout as well, while manuals on book design [Marshall] prescribe formulations for annotation and page sequencing.

The authors are not aware of a complete, generic taxonomy of graphic design together with examples of each entity in the hierarchy. Consequently, we formulated a taxonomy suitable for our project. It seeks to organize the ways in which visible language can be presented. The taxonomy is not exhaustive, nor is it rigorously systematic; however, it does provide visual examples for as many entries as possible. The subjects include treatments of individual elements (characters, words, lines), groups (paragraphs, pages), segments, zones, even entire documents. Particular attention is given to the treatment of character specification (including font family, size, spacing, weight, width, texture, and style) and line specification (orientation, spacing, justification, and function in the page).

By presenting typical variations of the visual elements of programs, it was easier for computer scientists and graphic designers to carry on a dialogue about the means and results of different strategies for visualization. In part this taxonomy was necessary because the typical computer scientist is unaware of the varieties of display and is ignorant of the technical terms for refering to these differences. A visually oriented taxonomy of visible language is an indispensable research tool for our research in computer graphics.

A Taxonomy of Constructs from the Programming Language C

In approaching the systematic mapping of C constructs onto typographic constructs, we first required a systematic description of the C language. We could find no existing description of C suited to our purposes, so we were forced to develop one. The result was neither a formal description of syntax and semantics, nor a reference manual, nor a tutorial introduction, but shares elements in common with each.

Details of the taxonomy are not relevant to this discussion. It suffices to say that we enumerated and classified all C constructs into tokens, identifiers and

declarations, expressions, statements, blocks, function definitions, program structuring tools, and C preprocessor constructs. With this schema, we could then begin the study of suitable visual representations of the language.

Folk Designs of C Programs

By folk designs of C programs we mean attempts by programmers with little or no background in typography and graphic design to improve the appearance of their programs. For the most part these alterations focus on regularizing the indentations of the component parts of PASCAL statements [Hueras, Grogono, Gustafson]. In a rare case some insight has been gained into the problems of effectively breaking lines within statements [Rose]. These improvements have not had a major impact upon the design of programs, in part due to the fact that the typographic environment is usually assumed to be that of a typewriter with fixed character-width. One individual has observed that regular indentation for nesting might allow the programs to appear without the numerous braces of compound statements [Leinbaugh]. Even with the limitations of fixed character-width and low-resolution typography, some systematic approaches to page layout, including program metadata and higher levels of comments are possible. However, the programmer's attention seems not to have focused on this aspect, but to have concentrated more upon the source code itself.

It was nevertheless reasonable and valuable to collect examples of informal approaches to graphically designing the source text of computer programs. These images serve as a starting point and remind us that programmers without professional skills in graphic design are aware of the need for better designs and are capable of evaluating differences among them even if they can not easily generate fundamentally new visual approaches on their own.

Systematic Designs of C Programs

At this point, we could begin systematic attempts to apply the full palette of graphic design techniques to reveal and express the meaning of C programs. We have subdivided the problem into six research areas:

1. *The Spatial Composition of Comments.* We explore in a systematic manner various methods for presenting program comments in relationship to program code. We emphasize a hierarchy of comments and a two column page format in which to express this hierarchy.
2. *The Typography of Tokens.* We explore in a systematic manner various mappings from C token attributes to typographic attributes.
3. *The Visual Parsing of Expressions.* We explore in a systematic manner various methods of using typographic attributes to enhance the ability of a reader to parse complex program expressions.
4. *The Visual Parsing of Statements.* We explore in a systematic manner various methods of using typographic attributes to enhance the ability of a reader to parse complex program statements. Particular attention is paid to brace styles, indentation rules, and techniques for breaking long lines.
5. *The Presentation of Program Structure.* We explore in a systematic manner various methods of enhancing the structure of a program in terms of its constituent parts, for example, its constituent declarations, statements, and function definitions. Particular use is made of typographic rules and spatial grids, headlines, and tables.
6. *The Presentation of Program Metadata.* We explore in a systematic manner various methods of enhancing the display of a program in relationship to the relevant data describing the context in which the program was created, is maintained, and is used.

We have constructed a simple visual C compiler which allows us to experiment in these six domains. Results of our early experiments are discussed briefly in the next section and will be presented in detail at the Conference.

Experimental Results

The accompanying figures show a preliminary example of automatically typeset pages. A visual C compiler scanned and parsed the program, and issued text processing commands based upon its analysis of the program.

The source code combined with the text processing commands were then input to a computerized typesetting system, which produced the sample pages.

The pages represent the beginning of a program and a portion of its continuation on a following page. Comments are distinguished from source code in part by distinctions of text type (sans-serif vs. serif), and in part by location (some comments appear in a narrow column to the side of the main column). Comments in large type appear on the page as headlines and subheads to identify primary sections of the program. Gray tone values are used to distinguish classes of text. The areas of gray value have been tested to ensure that highlighting by the use of gray can survive one or two generations of photocopying without making the type appearing within the gray bars illegible.

Note that we have established a page design which is intended to be useful for programs that may be written by persons with varying approaches to programming style. Individual parts might be systematically identified program modules which are organized according to user or machine tasks, or they might be files created by the programmer which merely group function definitions.

We have parameterized the tokens of C in such a way that it is a relatively easy task to alter the typographic specifications of these tokens. Consequently, it has been easy to generate many different possibilities of type selection for source text that distinguishes code from comments. In these figures we are distinguishing

tokens by changes in typeface (san-serif vs. serif), weight (medium and bold), slant (roman and italic).

In terms of metadata, the two pages present examples of the size, location, and content of repetitive items that a programmer needs on a frequent basis. Additional sketches have been designed for a title page, contents page, and metadata pages that relate to the generic page designs shown in the figures, but these additional sketches have not yet been automated.

Summary and Conclusions

Our research at this point is a modest inquiry into the manner in which graphic design specifications for visible language may clarify the structure and function of source code and comments. We have made progress in constructing an automatic means of realizing improved conventions for the display of source code and of facilitating experimentation with approaches to these conventions. Even within this limited scope, many interesting and valuable areas of further investigation have revealed themselves. Their exploration will require careful analysis, testing, and evaluation, as well as creative imagination in formulating new categories and concepts of program symbology and metadata. Among issues that seem worthy of further study are these:

What are relevant initial program metadata, and how should these be displayed?

What would a taxonomy of program comments reveal about the length, position, reference relationship, linguistic style, and literary style of these comments? What implication does this have for typographic presentation of these comments?

How can the visual C compiler easily be used to construct significant visual metrics of program structure? What kinds of visual indexes would be useful to see?

Is our schema for presentation general enough to account for varying views of program structure such as modules vs. files? Is the presentation equally suitable for the novice as well as the expert? Is the format appropriate for languages other than C? Is our design for a vertical page also adaptable to horizontal screen layouts?

At this stage we have accomplished a relatively small amount of software support for program visualization, yet we have in place already an armature with which many different alternatives may be constructed. This in itself is a valuable tool, for much of the future effort will be spent refining and carefully evaluating variations in type size, placement, symbolization, etc. Each of these small changes contributes to the success of the total graphic design.

We can conclude that enhanced program presentation can in fact be accomplished in an automatic manner with high quality phototypesetters. These may be used to deliver for the first time on a regular basis, beautiful and useful program presentations to the program builder, maintainer, and user.

References

Baker, F.T., Chief Programmer Team Management of Production Programming, *IBM Systems Journal* 11:1 (1972), 56-73.

Chaparos, Ann, *Notes for a Federal Design Manual,* Chaparos Productions, Washington, DC, 1981.

Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A.R., *Structured Programming,* Academic Press, London, 1972.

Deutsch, L. Peter and Taft, Edward A., "Requirements for an Experimental Programming Environment," *Xerox Palo Alto Research Center Report CSL-80-10,* June 1980.

Gerstner, Carl, *Compendium for Literates,* MIT Press, Cambridge, 1978.

Grogono, Peter, "On Layout, Identifiers and Semicolons in Pascal Programs", *SIGPLAN Notices 14:4,* 35-40.

Gustafson, G.G., "Some Practical Experiences Formatting Pascal Programs", *SIGPLAN Notices 14:9,* 42-49.

Gutz, S., Wasserman, A.I. and Spier, M.J., Personal Development Systems for the Professional Programmer, *Computer,* April 1981, 45-53.

Higgins, David A., *Program Design and Construction,* Prentice-Hall, Englewood Cliffs NJ, 1979.

Hueras, Jon and Ledgard, Henry, "An Automatic Formatting Program for Pascal", *SIGPLAN Notices 12:7,* 82-84.

Kernighan, Brian W. and Plauger, P.J., *Software Tools,* Addison-Wesley Publishing Company, Reading, 1976.

Leinbaugh, Dennis W., "Indenting for the Compiler", *SIGPLAN Notices* 15:5, 41-48.

Marcus, Aaron and Baecker, Ronald, "On the Graphic Design of Program Text", *Proceedings of Graphics Interface 82,* 1982, 302-311.

Marshall, Lee, *Bookmaking: The Illustrated Guide to Design and Production,* R.R. Bowker, New York, 1965.

Nassi, I. and Shneiderman, B., "Flowcharting Techniques for Structured Programing," *SIGPLAN Notices 8:8,* 12-26.

Oppen, D.D., "Prettyprinting," *ACM Transactions on Programming Languages and Systems* 2:4, October 1980, 465-483.

Organick, E. and Thomas, J.W., Computer-generated Semantics Displays, *Proc. IFIP Congress,* Applications Volume, 1974, 898-902.

Parnas, D.L., On the Criteria to be Used in Decomposing Systems into Modules, *Comm. of the ACM* 15:12 (December 1972), 1053-1058.

Rose, G.A. and Welsh, J., "Formatted Programming Languages", *Software -- Practice and Experience* 11, 1981, 651-669.

Ross, Douglas T., Structured Analysis (SA): A Language for Communicating Ideas, *IEEE Transactions on Software Engineering* SE-3:1, January 1977, 16-34.

Ruder, Emil, *Typographie,* Hastings House, Visual Communication Books, New York, 1973.

Teitelman, Warren, "A Display Oriented Programmer's Assistant," *Int. Jour. Man-Machine Studies,* 11, 1979, 157-187.

Wasserman, A.I., *Tutorial: Software Development Environments,* IEEE Computer Society Press, Los Alamitos CA, 1981.

Wirth, N., Program Development by Stepwise Refinement, *Comm. of the ACM* 14:4 (April 1971), 221-227.

Yourdon, E., *Structured Walkthroughs,* Prentice-Hall, Englewood Cliffs NJ, 1979.

Desk Calculator

Desk Calculator

```
#include <stdio.h>
#define MAXOP 20                /* Max size of operand, operator */
#define NUMBER '0'              /* Signal that number found */
#define TOOBIG '9'              /* Signal that string is too big */
```

This reverse Polish desk calculator adds, subtracts, multiplies and divides floating point numbers. It also allows the commands '=' to print the value of the top of the stack and 'c' to clear the stack.

Control Module

```
calc ()
{
    int type;                   /* Operation type */
    char s[MAXOP];              /* Buffer containing operator */
    double op2,                 /* Temporary variable */
           atof(),              /* Converts strings to floating point */
           pop(),               /* Pops the stack */
           push();              /* Pushes the stack */

    /* Loop while we can get an operation string and type */
    while ((type = getop(s, MAXOP)) ≠ EOF)
        switch (type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if (op2 ≠ 0.0)
                    push (pop() / op2);
                else
                    printf("zero divisor popped\n");
                break;
            case '=':
                printf("\t%f\n", push(pop()));
                break;
            case 'c':
```


calc, continued

```
                clecr();
                break;
            case TOOBIG:
                printf("%.20s ... is too long\n", s);
                break;
            default:
                printf("unknown command %c\n", type);
                break;
        }
}
```

Stack Management Module

```
#define MAXVAL 100              /* Maximum depth of val stack */
int sp = 0;                     /* Stack pointer */
double val[MAXVAL];             /* Value stack */
double push (f)                 /* Push f onto value stack */
double f;
{
    if (sp < MAXVAL)
        return (val[sp++] = f);
    else
    {
        printf("error: stack full\n");
        clear();
        return(0);
    }
}
double pop ()                   /* Pop top value from stack */
{
    if (sp > 0)
        return(val[--sp]);
    else
    {
        printf("error: stack empty\n");
        clear();
        return(0);
    }
}
clear ()                        /* Clear stack */
{
    sp = 0;
}
```