Understanding the Effects of Code Presentation

Jason T. Jacques Per Ola Kristensson

Department of Engineering
University of Cambridge, UK
{jtj21,pok21}@cam.ac.uk

Abstract

The majority of software is still written using text-based programming languages. With today's large, high-resolution color displays, developers have devised their own "folk design" methodologies to exploit these advances. As software becomes more and more critical to everyday life, supporting developers in rapidly producing and revising code accurately should be a priority. We consider how layout, typefaces, anti-aliasing, syntax highlighting, and semantic highlighting might impact developer efficiency and accuracy.

Keywords software visualization, syntax highlighting, programming

1. Introduction

Modern display technology offers practically unlimited presentation opportunities. Developers may have access to multiple large, bright displays with resolutions that may exceed human visual acuity [1]. Despite this flexibility, software developers, and some academics [3], have been resistant to significant change in programming environments. Programming is still largely a text-based activity.

Within this text-based environment, one way to provide additional support to programmers is to use secondary notation. Secondary notation does not change the formal meaning of the code but allows additional context to be layered into the code [5]. Examples of secondary notation in source code editors include syntax highlighting, and indentation. Developers can benefit from decreased cognitive load while reading, writing, and rewriting code [14]. Programmers have embraced this facility to add additional meaning to their work [2] by including comments, using indentation styles, and applying syntax highlighting.

2. Background

Understanding and improving the process of reading text on a display is not a problem unique to software developers. Optimizing the presentation of symbols on displays is essential to maximize accuracy and minimize discomfort. Early work by Radl [10] investigated how green, white and yellow phosphors used in contemporary VDUs (visual display units) affected transcription error rates. The author concludes that around the optimal wavelength of 555 nm, brightness and contrast take over as the significant determinant of accuracy. Radl also notes that color preference "seems to be caused mainly by psychological factors rather than by physiological mechanisms" [10].

The limitations of early display technologies lead to interest in typesetting computer code to improve readability. Baecker and Marcus [2] demonstrated their *visual C compiler*, leveraging the higher resolutions offered by laser printing. Their tool systematically applies typographic styles and annotations to source code to improve both the aesthetics and usefulness of the source code. Expecting the output to be duplicated, the authors ensured that gray values used for highlighting would maintain the integrity of the text after multiple generations of photocopying.

Shneiderman [13] describes his vision for a model programming environment, which includes much of the functionality depended on by developers today, such as syntax highlighting, code folding, macros, and version control. Shneiderman [13] presents his model environment without formal validation as a foundation to stimulate future work and discussion of human factors in software development.

With the increased flexibility of modern computing, augmenting and enhancing the systematic presentation of text in the editor offers further opportunities to improve developers' reading and comprehension of code [13]. By using both controlled studies and non-invasive capture of real-world developer practice [15], usage patterns can be discerned to discover and develop targeted enhancements. We believe such improvements have the potential to increase development speed and reduce errors in the resulting software.

3. Hypotheses

A variety of advances have been made in the presentation of onscreen code. However, much current practices lacks validation. This is highlighted by the numerous opinions and anecdotes we describe in this section. Websites and blogs, and the editors themselves, are a modern "folk design" resource. What is now needed is verification and formalization of this untested corpus of potential improvements. The following considers the impact of a variety of these non-invasive changes to the editor environment on the developer.

3.1 Layout and Indentation

The use of indentation (tabs or spaces to visualize scope) and layout (other intentional positioning, such as grouping of statements to infer related meaning) to enhance or add additional meaning to code is common among developers. Leinbaugh [6] points out the duplicate effort by programmers to maintain appropriate visual style with indentation and accurately represent the actual or preferred scope of code. Despite this maintenance effort, Miara et al. [7] demonstrate support for using limited indentation to aid comprehension, with larger indentation reducing or eliminating the benefit.

Layout can also be used to suggest association or disassociation between groups of program statements. Green and Petre [5] suggest these "paragraphs" (visual grouping of code to inply an association) and the use of "rhyme" (repetitive code structures to highlight intentional or unintentional differences) aid in understanding code. For developers, however, the final layout and indentation are part of the creation process, and can be a very personal matter, engendering strong feelings[1,2].

Others have taken these informal practices and attempted to improve upon and automate them. The investigation of "folk design" by Baecker and Marcus provided a starting point for their own work on their *visual C compiler* [2]. Similarly, the *book format paradigm* by Oman and Cook [9] uses tools to automatically apply indentation and layout changes to the source code. Focusing on practical access to the code, their work extends the layout by adding a preface, table of contents, and other typographic features. These changes aim to provide identifying and organizational cues to the programmer.

H1 *Limited, consistent indentation assists developer comprehension.*

H2 *Grouping and ordering of program statements can assist developers in identifying patterns and errors.*

3.2 Typefaces and Fonts

To read code, the typeface used to render it must be legible. Even basic editors, such as Windows Notepad, offer the ability to change the typeface. The selection of appropriate typefaces is much discussed in the online developer community[3,4,5]. Among developers monospace seems to be a foregone conclusion, with idealized requirements often omitting to specify this explicitly[3,4]. Even when developers consider proportional fonts, this is often quickly dismissed as a non standard presentation[6].

Microsoft *Small Basic*, a programming environment aimed at beginners [11], elects to use both monospace text for executable code and a proportional italicized font for comments. This alternative presentation of the documentary parts of the code aids in the visual segmentation and offers an interesting alternative to the monospace monoculture.

While many editors limit the editor window to just one typeface, most offer additional differentiation options using bold, italicized, and underlined variants. The use of alternative fonts for certain syntax suggest the increased significance of these elements [13] and can impart additional meaning to the following code.

H3 *Different typefaces aid developers in segmenting code into component parts, such as program instructions and comments.*

H4 *Varying the font, such as using boldface or italic, assists developers in tokenization of syntax.*

3.3 Anti-aliasing

Reading on a display is a different experience than reading on paper. While computers display a rigid grid of pixels at a fixed resolution, the printed word is smooth, and sharply rendered. This differing presentation can affect reading speed. Experiments by Gould [4] show that much of the performance loss caused by reading from a screen, can be negated by having it more closely resemble paper-based renderings, including the use of anti-aliasing. Anti-aliasing allows the jagged, stair-stepping seen in computer graphics and text to be smoothed out using intermediate coloration in some adjacent pixels.

The use of anti-aliasing of text has been controversial among developers with some describing the effect as "muddy"[6] and "extremely fatiguing"[7], even for typefaces specifically designed for use with anti-aliasing. This may be impacted by developer inclinations to use small point sizes to maximize the amount of text shown. As text becomes smaller a larger proportion of the rendered character is com-

[1] J. Zawinski, *Tabs versus Spaces*, 2000, http://www.jwz.org/doc/tabs-vs-spaces.html

[2] J. Atwood, *Death to the Space Infidels!*, 2009, http://blog.codinghorror.com/death-to-the-space-infidels/

[3] T. Lowing, *Monospace/Fixed Width Programmer's Fonts* http://www.lowing.org/fonts/

[4] J. Atwood, *Programming Fonts*, 2004, http://blog.codinghorror.com/progamming-fonts/

[5] D. Benjamin, *Top 10 Programming Fonts*, 2009, http://hivelogic.com/articles/top-10-programming-fonts

[6] R. Strahl, *Clear Type this is supposed to be better?*, 2005, http://weblog.west-wind.com/posts/2005/Jul/25/Clear-Type-this-is-supposed-to-be-better

[7] J. Atwood, *Consolas and ClearType*, 2005, http://blog.codinghorror.com/consolas-and-cleartype/

prised of smoothing shaded pixels, giving the character a more blurry overall appearance.

As screen resolutions have improved and display sizes increased anti-aliased fonts have become more acceptable[5]. Higher resolutions have allowed for smaller, less perceptible pixels, minimizing the ability to discern individual modifications to the letter outlines. Similarly, larger display sizes allow more computer code to be displayed simultaneously and lessen the need to maintain small point sizes to show large amounts of text.

H5 *Anti-aliasing improves onscreen code readability at larger point sizes.*

3.4 Syntax Highlighting

Most professional code editors support some form of syntax highlighting. By differentiating certain features of the syntax, code editors assist in the tokenization of the code for easier parsing by the human reader [13].

At the time of writing, *Studio Styles*, a popular color scheme repository for Visual Studio, has over 3,000 syntax highlighting schemes available for download and use [8]. The most popular scheme, *Son of Obsidian*, combines a dark background for high-contrast with many of the same colors that Radl [10] found most beneficial for symbol recognition on dark screens. The *Son of Obsidian* scheme has over 500,000 downloads; twice as many as the next most popular.

Developers have attempted to formalize the color schemes they use. The *Solarized* project by Ethan Schoonover [12] is one such example. Solarized uses a lower contrast color palette to reduce fatigue, while maximizing differentiation between syntactic elements using complimentary colors (e.g, red-green, yellow-purple, etc.). Solarized also allows the background color of the editor to be switched from light to dark while preserving the contrast relationships with minimal color changes between modes. This allows the most appropriate background to be used for the ambient light level.

There has also been interest in scientific testing of color schemes and disappointment among developers[9,10] by the dearth of available publications. While some developers seem skeptical of an objectively "better" color scheme[9], Radl [10] presents an example of a physiological phenomenon that might be exploited to this end (namely, varying color sensitivities of the eye).

H6 *Color schemes which tokenization syntax by exploiting physiological properties aid readability of code.*

H7 *Very high contrast color schemes can induce eye fatigue over long periods of use.*

[8] Studio Styles, *Visual Studio Color Schemes*, https://studiostyl.es/

[9] Programmers StackExchange, *Syntax-highlighting color scheme studies*, 2011, http://programmers.stackexchange.com/questions/89936/syntax-highlighting-color-scheme-studies

[10] Skeptics StackExchange, *Are light-on-dark colour schemes for computer screens better for programmers?*, 2011, http://skeptics.stackexchange.com/questions/6925/are-light-on-dark-colour-schemes-for-computer-screens-better-for-programmers

```c
#include<stdio.h>
#include<stdlib.h>

const double pi = 3.14159265;

int main(int argc, char *argv[]) {
    double radius, diameter, circumference, area;

    if (argc < 2) return -1;
    radius = atof(argv[1]);

    diameter = 2 * radius;
    circumference = pi * diameter;
    area = pi * (radius * radius);

    printf("Diameter     : %f\n", diameter);
    printf("Circumference: %f\n", circumference);
    printf("Area         : %f\n", area);

    return 0;
}
```

Figure 1. Example of *Semantic Highlighting* in KDevelop. Note that variables names each have their own color and retain this throughout their scope to aid identification.

3.5 Semantic Highlighting

With the perceived benefits of syntax highlighting, there has been interest in expanding the scope from syntactic elements to semantic meaning of code. Eclipse 3.0, a popular Java-based editor and IDE, introduced *advanced highlighting*[11], styling local variables, an objects fields, and static fields separately. This separation of constants from variables reduces developer dependency on more manual stylistic techniques, such as using uppercase names, to indicate immutability.

In 2009, Nolden [8] coined the term "semantic highlighting" for his work on KDevelop. While syntax highlighting uses a deterministic parser to aid in tokenizing syntactic elements, semantic highlighting attempts to reveal the meaning of the code. One of the earliest implementations of semantic highlighting first appeared in *KDevelop* [8]. Like Eclipse, KDevelop highlights items based on scope, however it additionally applies local variable colorization. By assigning individual variables their own color the developer can trace how data flows and is manipulated without actually reading the text names given to each variable. See Figure 1.

Recently, renewed interest was given to semantic highlighting by its rediscovery by Evan Brooks[12]. Brooks' rediscovery encouraged discussion on the concept and with some developers producing plugins for popular languages and IDEs to support the semantic highlighting[13]. Dynamic coloration requires much more flexible color schemes and can reduce the uniqueness of hues applied to traditionally highlighted syntax [13]. Brooks initial implementation attempted to evenly distribute variables over the spectrum[12],

[11] IBM, *Eclipse 3.0 News - Part 4*, 2004, http://archive.eclipse.org/eclipse/downloads/drops/R-3.0-200406251208/eclipse-news-part4-R3.html

[12] E. Brooks, *Coding in Color*, 2014, https://medium.com/@evnbr/coding-in-color-3a6db2743a1e

[13] Reddit, *Coding in color : programming*, 2014, https://www.reddit.com/r/programming/comments/1w76um/coding_in_color/

however exploiting physiological properties of human vision to better differentiate tokens might offer improvements.

Some developers voiced concern that semantic highlighting might overload the presentation and further obscure the meaning of the code. One observer commented that semantic highlighting might "turn your code into a confusing christmas tree."[13] Nolden anticipated initial developer reluctance to use semantic highlighting, and included an option to disable the feature, commenting that "the best thing about it: You dont have to use it at all." [8]

H8 *Highlighting differing scope aids developer identification of the impacts of changes to variables.*

H9 *Semantic highlighting improves developer understanding of data flow through code.*

4. Discussion and Conclusion

While the presentation of code has undergone a number of changes over the last three or four decades, progress has been slow. Both developers and academics have attempted to improve the editing experience through a variety of techniques. However, without specific design insights it can be difficult to transform amateur practice into professional formalism [5]. Differing opinions[1,2] and resistance to change[6,7,12,13] has hampered progress. However, some enhancements such as syntax highlighting have, at least conceptually, been almost universally embraced[8,12,13] .

Despite promising early work [4, 10] pointing to physiological mechanisms that could be exploited to enhance on-screen reading, the slow adoption of technologies such as anti-aliasing[6,7] by developers has hampered their application in code editors. With the widespread availability of large, high-resolution, color displays the dearth of research investigating the presentation of code is disappointing[9,10].

Developers have instead independently pursued formalization [12] and improvement [8] of their editing environments. It is clear that untested theories and the personal preferences of those creating editors have become the state of the art. Now, it is essential that rigorous study of these behaviors is brought up-to-date with the changes in technology that have been seen in the intervening decades since human factors in programming was under serious study.

This paper has revisited some early work to improve the developer experience of editing code. We have considered how changes to the editor have been rejected, adopted, and embraced by developers. We suggest changes that might be tested and refined to improve readability of code. By focusing only on the text itself we ignore the impact that other innovations, such as integration and automation of the build process or live-state inspection and steppable debuggers might have had on developers.

However, with software so critical to everyday life, a single error can impact millions of direct and indirect users. As writing code with text-based languages remains the predominant method used to develop software, focusing on this key interaction space is critical to aiding developers in their understanding of code and minimizing errors in the software we use everyday and that affects us all.

References

[1] H. P. Apple, T. K. Leonard-Green, E. M. Harvey, J. M. Miller, and D. Apple. Suitability of computer generated grating acuity stimuli for assessment of grating acuity in children. *Investigative Ophthalmology & Visual Science*, 55(13):2738–2738, Apr. 2014. ISSN 1552-5783.

[2] R. Baecker and A. Marcus. On Enhancing the Interface to the Source Code of Computer Programs. In *Proc. CHI '83*, pages 251–255, New York, NY, USA, 1983. ACM. ISBN 0-89791-121-0.

[3] E. W. Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32:1398–1404, 1989.

[4] J. D. Gould, L. Alfaro, R. Finn, B. Haupt, and A. Minuto. Reading from CRT Displays Can Be as Fast as Reading from Paper. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 29(5):497–517, Oct. 1987. ISSN 0018-7208, 1547-8181.

[5] T. R. G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages & Computing*, 7(2):131–174, June 1996. ISSN 1045-926X.

[6] D. W. Leinbaugh. Indenting for the compiler. *ACM SIGPLAN Notices*, 15(5):41–48, 1980.

[7] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program Indentation and Comprehensibility. *Commun. ACM*, 26(11):861–867, Nov. 1983. ISSN 0001-0782.

[8] D. Nolden. C++ IDE Evolution: From Syntax Highlighting to Semantic Highlighting, Jan. 2009. URL https://zwabel.wordpress.com/2009/01/08/c-ide-evolution-from-syntax-highlighting-to-semantic-highlighting/.

[9] P. W. Oman and C. R. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, 1990.

[10] G. W. Radl. Experimental investigations for optimal presentation-mode and colours ofsymbols on the CRT-screen. In *Ergonomic aspects of visual display terminals*, pages 127–135. Taylor & Francis, 1980.

[11] V. Raji. *Microsoft Small Basic: An introduction to Programming*. Microsoft, Jan. 2013.

[12] E. Schoonover. Solarized, 2011. URL http://ethanschoonover.com/solarized.

[13] B. Shneiderman. A Model Programming Environment. In H. R. Hartson, editor, *Advances in human-computer interaction.*, volume 1, pages 105–131. Ablex, Norwood, N.J., 1985. ISBN 0-89391-244-1.

[14] J. Sweller. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, 4(4):295–312, 1994. ISSN 0959-4752.

[15] Y. Yoon and B. A. Myers. Capturing and Analyzing Low-level Events from the Code Editor. In *Proc. PLATEAU '11*, pages 25–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1024-6.