Software Engineers are Human Too

Michael L. Van De Vanter

December 15, 1993

Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue, MTV29-112
Mountain View, CA 94043-1100
Michael.VanDeVanter@Eng.Sun.COM

1 Introduction

Software engineers increasingly understand that human-computer interaction issues are essential to good software design. Like the proverbial cobbler's children who want for shoes, however, our own tools receive far too little of that attention.

Language-based editing systems represent an important evolutionary thread in our tool development. These systems allow engineers to create, browse, and modify software documents in terms of the formal languages and notations in which they are written (for examples in terms of "statements," "integer expressions," and "assignments with type-compatibility problems") not just in terms of their superficially textual characteristics. However a lack of widespread acceptance has proven something of a disappointment to those who envision the potential contribution of these systems.

Part of the problem has been a lack of language-based technology appropriate for interactive use, in contrast to the much better understood world of batch-oriented program compilation. Several generations of experimental language-based editing systems have made significant progress with the technology [Bahlke 86][Borras 88][Donzeau-Gouge 84] [Notkin 85] [Reps 84], and practical systems of this kind appear within reach.

Experience with these almost-practical systems suggests that many problems remain, for example characterized by:

1. Annoying restrictions on text-based editing, the style of interaction most people prefer.
2. Failure to address the real productivity bottleneck: the difficulty people have understanding programs.
3. Exposure of underlying language-based technology to people, leading to inappropriate user models of system state and document structure.
4. Monolithic document presentations that fail to exploit all the information available in the programming environment.
5. Brittle system behavior in the presence of ill-formed programs or inconsistent system information.
6. Awkward (or missing) mechanisms for incorporating new languages.
7. Inflexibility because of closed data models and weak extension facilities.

Systems that address some of these issues typically do so at the expense of others. These usability problems go far beyond the superficial graphical user interface design issues such as the arrangement of menus and the appearance of buttons. At issue are questions about how software engineers work, what tools they already know and use, how they understand the notation, and what (human) performance bottlenecks might profitably be addressed by language-based editing systems.

Recent work carried out as part of the *Pan* project at the University of California Berkeley [Ballance 92] [Van De Vanter 92b] revisits the design of these systems by posing user-centered rather than technology-centered questions, with results that have implications on the following issues [Van De Vanter 92a]:

- internal software architecture;
- services offered to users;
- configuration mechanisms;
- styles of interaction;
- integration with other tools in the working environment; and
- the suitability of current language-based technology for the challenge.

The thesis of that investigation was that the success of language-based editing systems has been limited by inattention to user-centered issues concerning the context in which the tools are needed. It was further argued that the design of a language-based editing system presents a *user interface problem*, not only between user and system (in the conventional sense of the term) but between software engineer and software documents (the more important meaning in this context).

2 A Framework for User Interaction

With the design challenge posed in this way, new questions could be asked and new balances struck on fundamental choices such as text- versus structure-based interaction, language-specific versus generic services, and frequent versus infrequent analysis. The application of *user-centered system design principles* [Norman 86] casts new light on the issues and suggests that a well-founded, principled, and coherent approach to the design of language-based editing systems is possible.

The *Pan* framework for user interaction begins with a basic separation of design concerns not evident in earlier systems:

1. internal document representation and analyzer implementation;
2. configurable, language-independent mechanisms to support user interaction with software documents;
3. a coherent and flexible set of user-visible system features and policies; and
4. adaptation of the system to particular working contexts.

Pan I version 4.0 is a fully implemented, multilingual, language-based editing and browsing system [Ballance 92] [Downs 91] [Van De Vanter 92b] that embodies this framework and demonstrates the viability of the approach.

3 Guiding Principles

An important goal of *Pan*'s design framework is to “decouple” user interaction in these systems from the linguistic and implementation details of their enabling technologies, and in particular from the compiler-oriented approach that has dominated their design in the past. Software engineers do not think of programs in the same terms that are useful for compiler designers. For example, a person confronted with a new program first reads comments and then examines the names of program entities, for example procedures and variables. In contrast, a compiler fed the same program first discards comments and then abstracts away all names. This example is not meant to imply that a language-based editing system should analyze comments, but it does suggest the depth of the conceptual challenge for designers who seek appropriate application of the technology.

Adaptation to working contexts is captured in *Pan*'s design framework by the notion of multiple *view styles* for user interaction, each specialized for a particular combination of user population, task at hand, and language being used. A view style

1. includes traditional syntax and static-semantic language descriptions, but may extend to extra-lingual analysis such as stylistic and usage guidelines;
2. specifies services to be provided and specializes generic services for the particular language;
3. defines a visual context, including typography and use of color; and
4. configures details of interaction, including keystroke and menu-bindings.

A human designer creates view styles. A working *Pan* system includes a suite of view styles that collectively offers appropriate services and uniform user interaction. *Pan*'s design framework provides tools, guidelines, and examples, among which are solutions to usability problems that plague earlier generations of systems.

Finally, *Pan*'s design framework is *open*. To realize the full power of language-based interaction, the editor must function as an interface through which an open-ended collection of language-related services can be delivered to software engineers. Known as *applications* in the *Pan* framework, these additional services can be added to *Pan* using its extension language, rich configuration mechanisms, and an extensible data repository. Alternately they can be delivered by integration with other tools, for example allowing *Pan* to serve as a user interface for compilers, profilers, debuggers, and code auditors.

4 Accomplishments

The *Pan I* prototype addresses each of the usability problems identified above and continues to support ongoing research at Berkeley and elsewhere; current topics include advanced software viewing and browsing, code optimization and generation, reverse engineering, and static-semantic analysis. Some of *Pan*'s technology is being carried forward into *Pan*'s successor at UC Berkeley, the *Ensemble* project [Graham 92].

Several novel aspects of *Pan*'s design, developed while meeting these goals, deserve mention.

Isolation of Language-Based Technology: It is tempting think of language-based editing systems as interactive compilers, but language-based technology developed for compilers ports badly into the domain of user interaction. This proved to be true even in *Pan* where the project began with the benefit of insight from two earlier generations of language-based systems. *Pan*'s layered design model separates language-based analysis mechanisms from user-oriented, language-independent services; most of the system's design accommodates user-centered design choices without excessive coupling to the batch-oriented, compiler model of software structure.

Operand Class Abstraction: A description-driven mechanism in *Pan*'s language-independent kernel drives a variety of user-oriented services, ranging from simple navigation to complex projections in alternate views. The abstraction solves several problems in user interface design and permits services to be adapted for uniform operation across multiple language-based view styles.

Gracious Services Metaphor: Frequent inconsistency between edited text and analysis-derived data is inescapable in *Pan*, and will persist in any similar system that scales up to confront large-scale propagation of changes. An appropriate design metaphor (as well as some experimental implementations) leads to services that continue to be useful when operating with approximate information.

Elements of User Interaction: *Pan* users see a simple system through which an open-ended variety of potentially complex information may be exploited. Simplicity derives from a few basic services that can be applied in a variety of ways, but which have simple and predictable behavior of their own.

Smart Services Metaphor: *Pan*'s structure-oriented commands are presented as optional, better-informed elaborations of familiar text-based commands, avoiding the confusion that can arise from a separate command set based on unseen structure.

Coherent Interaction with Document Structure: A view style specification describes an interface, implemented by *Pan*, between users and documents in a particular language. Each interface can be tailored for particular users, their tasks, and an underlying language. Much of the richness and effectiveness of *Pan* derives from view style design.

The View Style Designer: Formal language description is not an adequate basis for specifying user interaction. A tool like *Pan* embodies a complex relationship among (a) users, (b) the medium in which they work, and (c) the tasks they perform; it must be *designed* with these factors in mind, a challenging task. The *Pan* system cannot guarantee good design; it offers a framework, building blocks, examples, and guidelines that enable good design.

5 Open Issues

Work on the framework for user interaction in *Pan* leaves open a number of issues.

User Experience: More empirical evidence is needed to validate and refine the user interaction techniques developed here. This kind of experience can only be gathered by experimenting with a flexible system such as *Pan* in production environments using production languages.

Advanced Visual Presentation: Many potentially useful presentation techniques, based on the static book publishing paradigm, are not supported by *Pan I*'s prototype rendering engine. Just as batch-oriented compiler technology doesn't necessarily port well into an interactive environment, however, some design choices made for a static publishing paradigm may not be appropriate in a more dynamic context, and some techniques may not justify their implementation costs.

Integration with Other Tools: *Pan* mechanisms for viewing software documents are designed to exploit a wide variety of possibly large scale information. *Pan*'s potential will only be realized through integration with other tools one expects to find in a modern computer-aided software engineering environment: more ambitious analyzers (data flow for example), debuggers, profilers, test coverage generators, design documentation systems, and persistent storage.

Object-Oriented Programming: Much of the experience and insight that drove *Pan*'s design predates widespread acceptance of object-oriented design and languages. These languages are still in flux, and only the most tentative results are starting to appear that will cast light on the cognitive processes of programmers working in the new design paradigm. Many of *Pan*'s techniques will apply, but new ones will probably be needed to accommodate changing notions of system modularity and connectivity.

Language Extension: *Pan*'s language description and analysis model is not well suited to languages with powerful extension facilities, for example the macro processing facilities supported by Commonlisp. Closely related is the delivery of services that effectively blur the boundary between language definition and editing system. *Pan*'s techniques for user interaction should apply in most cases, but they may need to be adapted (as the language analysis model must change) for the more dynamic context.

Language-Based Technology: Technology that can be shared between language-based editing systems and compilers must be developed and exploited in order to avoid the kinds of infrastructure problems discovered during this work. In the best cases, the boundary between the two applications will become blurred (as it will between editing systems, compilers, and their underlying languages). But it will not succeed until each component of the technology is recast into this new, more general role.

6 References

[Bahlke 86] Rolf Bahlke and Gregor Snelting, The PSG System: From Formal Language Definitions to Interactive Programming Environments, *ACM Transactions on Programming Languages and Systems* 8,4 (October 1986), 547-576.

[Ballance 92] Robert A. Ballance, Susan L. Graham and Michael L. Van De Vanter, The *Pan* Language-Based Editing System, *ACM Transactions on Software Engineering and Methodology* 1,1 (January 1992), 95-127.

[Borras 88] P. Borras, D. Clemént, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual, “CENTAUR: the system”, *Proceedings ACM SIGSOFT ‘88: Third Symposium on Software Development Environments*, November 1988, 14-24.

[Downs 91] Laura M. Downs and Michael L. Van De Vanter, “Pan I Version 4.0: An Introduction for Users”, 91/659, Computer Science Division, EECS, University of California, Berkeley, August 1991.

[Donzeau-Gouge 84] Véronique Donzeau-Gouge, Gérard Huet, Giles Kahn and Bernard Lang, “Programming Environments Based on Structured Editors: The MENTOR Experience”, in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe and Erik Sandewall (editors), McGraw-Hill, New York, NY, 1984, 128-140.

[Graham 92] Susan L. Graham, Michael A. Harrison and Ethan V. Munson, “The Proteus Presentation System”, *Proceedings ACM SIGSOFT ‘92: Fifth Symposium on Software Development Environments*, December 1992, 130-138.

[Norman 86] Donald A. Norman and Stephen W. Draper (editors), *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[Notkin 85] David Notkin, The GANDALF Project, *Journal of Systems and Software* 5,2 (May 1985), 91-105.

[Reps 84] Thomas Reps and Tim Teitelbaum, The Synthesizer Generator, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 19,5 (May 1984), 42-48.

[Van De Vanter 92a] Michael L. Van De Vanter, “User Interaction in Language-Based Editing Systems”, UCB/CSD-93-726, Ph.D. Dissertation, Computer Science Division, EECS, University of California, Berkeley, December 1992.

[Van De Vanter 92b] Michael L. Van De Vanter, Susan L. Graham and Robert A. Ballance, Coherent User Interfaces for Language-Based Editing Systems, *International Journal of Man-Machine Studies* 37,4 (1992), 431-466.