SML 94-0290

Practical Language-Based Editing For Software Engineers

Michael L. Van De Vanter

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

Abstract. Language-based editing systems have the potential to become a practical, central, and powerful part of every software engineer's toolkit, but progress has been limited by inattention to user-centered design issues. Major usability requirements for such systems include familiar, unrestricted text editing; coherent user interaction with software; rich, dynamic information display; multiple alternative views; uninterrupted service in the presence of ill-formedness, incompleteness, and inconsistency; description-driven support for multiple languages; and extensibility and customizability. Solutions require better understanding of software engineers and their tasks, appropriate design metaphors, new architectural organizations, and design for adaptation and extension.

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. Software engineers are human and they interact with computers; they deserve the best tools we can build.

Language-based editing systems represent an important advance in software engineering technology. These tools enable engineers to create, browse, and modify software documents in terms of the formal languages and notations in which they are written (for example in terms of "statements," "integer expressions," and "assignments with deprecated type conversions") not just in terms of their superficial 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 interactive language technology [BS86] [BCD+88] [DHK+84] [Not85] [RT84], and practical systems of this kind now appear within reach.

Experience with these almost-practical systems suggests that usability problems remain, problems that 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

Appears in *Software Engineering and Human-Computer Interaction: ICSE '94 Workshop on SE-HCI: Joint Research Issues, Sorrento, Italy, May 1994*, Proceedings, Lecture Notes in Computer Science vol. 896, Richard N. Taylor and Joelle Coutaz (editors), Springer Verlag, Berlin, 1995, 251-267

profitably be addressed by language-based editing systems. Solutions demand new design thinking on both user-visible behavior and underlying architecture.

Recent work carried out as part of the *Pan* project at the University of California Berkeley [BGV92] [VGB92] revisited the design of these systems by posing user-centered rather than technology-centered questions, with results that have implications on the following issues [Van92]:

- 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.

These user-centered questions begin with the intended context for such systems, discussed in Section 2, followed by a more specific discussion of requirements in Section 3. Section 4 introduces the research prototype developed during the project. Sections 5 and 6 describe design solutions, first with respect to how users are expected to understand the system, and second with respect to its internal organization. Section 7 summarizes and mentions open issues.

2 Context

Language-based editing systems must move far beyond the general-purpose, standalone text editors now widely used for programming. Figure 1 suggests their potential role for software engineering: as front ends to databases of design information, much like the Computer Aided Design (CAD) systems of other engineering disciplines. This presents two design challenges for user interaction. The conventional (difficult) problem is to build a user interface that makes the system usable. The larger (more difficult) challenge is to build an effective interface between software engineers and the underlying software documents. In this much more important sense, *the entire system is the crucial user interface* for software engineers.

The skills and expectations of software engineers are key considerations. These users can be presumed fluent with one or more conventional text editors (loathe to learn a new editor), fluent with one or more primary languages (checking for syntax errors does not add much value except when learning new languages), and fluent with other tools that deal with software documents (compilers, debuggers, and the like).

3 Requirements for Usability

Design requirements for language-based editing systems must address both the demands of the software engineering task milieu and the usability problems seen in past systems.

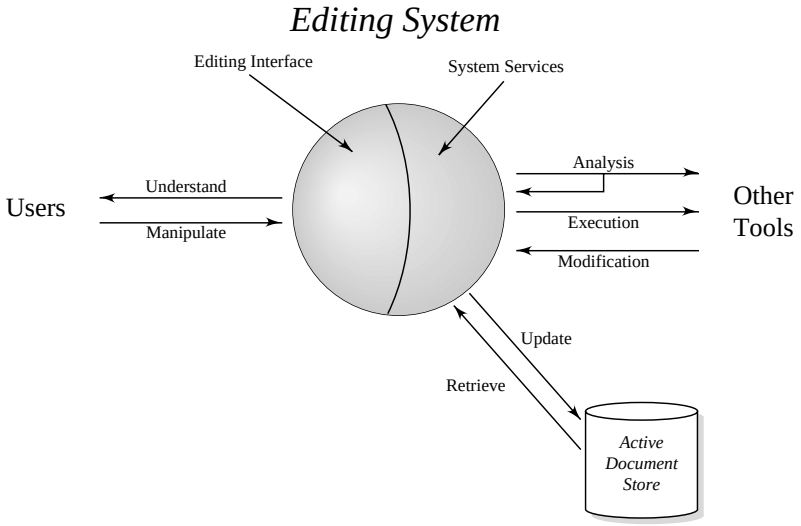

Figure 1: Editing system in relation to the programming environment

3.1 Familiar, unrestricted text editing

Language-based editing systems of an earlier generation were based on what will be called here the "the structural hypothesis," best expressed as "Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint. The Cornell Program Synthesizer *demands* a structural perspective at all states of program development" [TR81] (emphasis added). The structural hypothesis is flawed in principle and has not been confirmed in practice.

Demanding a structural perspective fails because the editing model (what the user can do, based on an internal tree) clashes with the presentation model (what the user sees, a field of text). It imposes on the user the cognitive overhead of understanding a complex, unseen relationship. Consequences of this failure appear in many forms, for example:

- A manual warns about a sequence of operations whose result is likely to surprise users, a result only comprehensible through a subtle line of reasoning involving the structure cursor's placement in the (invisible) internal tree [RT87, page 91]. It is an equally subtle problem to discover the sequence of operations that produces the intended results.
- Apparently sensible cut and paste operations can fail in ways that require heroic efforts to explain or repair [GL88] [Ler92].
- Moving between two structures adjacent on the screen may require a complex tree traversal, as these structures might be only be distantly related [CMP91].
- "... it is particularly inconvenient for editing text/program fragments that are non-structured (strings, comments) or poorly structured (expressions)" [Lan86].

- "It is not possible to insert or change text at an arbitrary point..." [BS92].

Many system designers retreated into hybrid approaches in which structural fragments may be edited textually in limited circumstances. Minör's attempt to correct the problem without abandoning the structural hypothesis led away from a text-oriented interface entirely for the SbyS program editor [Min90].

Other evidence, in addition to lack of commercial success, confirms that people will not accept restrictions to familiar text-oriented interaction [Nea87]. Much of software engineering notation is textual, and people are simply accustomed to it.

3.2 Coherent user interaction with software

Although text-oriented interaction must not be sacrificed, structure-oriented operations are crucial to exploiting the potential power of language-based systems. But what structure will make those operations coherent?

Earlier generations of language-based editing systems failed by driving user-accessible interaction directly from syntax trees, data structures that have more to do with the underlying technology than they do with how software engineers work. One symptom is that a syntax tree representation for a language is not unique; it reflects a set of implementation choices based only loosely on a formal language definition. Although the tree representations used by some editors are more "abstract" than others, and therefore allegedly more "natural" for users, tree representations are in practice designed to meet the needs of tool implementations (parsers, analyzers, and the like) and not those of coherent interaction with people.

Even if there were a canonical structural representation, it would not suffice. Syntactic structure is a useful "backbone," but software engineers manage language-related information that goes beyond the purely syntactic. The true structure of software is complex, multifaceted, and non-local. Different users and tasks require different uses of structure and different forms of access to the information within documents. Although the information must be broad in subject domain, it need not be deep (in the sense that program *plans* [LS86] [SE84] and *clichés* [RW88] are deep) to be useful.

Structural interaction with users must be flexible, able to accommodate any kind of "structure" for which sufficient information is available: lexical, syntactic, static-semantic, data-flow analysis, stylistic analysis, performance results, and anything else that software engineers need.

3.3 Rich, dynamic information display

Many language-based editing systems were designed to conserve keystrokes and prevent syntax errors, but the productivity bottleneck lies elsewhere. Software engineers spend far more time trying to understand, modify, and adapt software documents than they do creating them in the first place [Gol87] [Win79].

High-quality visual design and typography enhances comprehension of natural language documents, and recent studies suggest the same potential benefits for programs displayed on paper [BM90] [OC90]. This approach helps, but it must be adapted to the perceptual characteristics of CRT displays and to the dynamics of interaction.

Additional information (meta-information) must be added to the textual display dynamically as needed, using typographical variations such as type, color, and back-

ground, as well as elision and annotation. Evidence suggests that reading software is a cognitively active process and has a fine-grained task structure [KR91]. The reader repeatedly forms hypotheses, which are then confirmed or denied opportunistically by further reading, using a variety of information and reasoning strategies, depending on the information available [Let86]. Even when writing, programmers spend most of their time reading what they've just written. Writing software is a creative design process, and like many kinds of design it is done iteratively, with cycles of explicit interaction and feedback from what has been committed to notation so far [Joh85] [Sch83]. Software engineers constantly ask "Where am I now?" "What are the implications of what I've done so far?" and "What's left to do?" A language-based editing system must be ready with the right information at the right time.

3.4 Multiple, alternative views

Mark-up with meta-information is only one way to exploit the information available; in some cases, reorganization and filtering are more appropriate. For example tables of contents and indexes assist document comprehension without adding new information. Given the multiplicity of structural aspects present in software systems, and the variety of meta-information that can be produced, for example by data-flow analyzers and performance profilers, the range of potentially helpful views is large. A language-based editing system must be able to create such alternative views as needed.

3.5 Uninterrupted service despite "I3"

A persistent and general problem with language-based tools is that they fail to degrade gracefully in the presence of ill-formed, incomplete, or inconsistent information, described here as the "I3" conditions. In practice these are the natural states for documents under development, conditions where the software engineer is most likely to need help. A language-based editing system must not fail to deliver what service it can under these circumstances.

I1: Ill-Formedness. Software documents being modified are often at variance with an underlying language definition. Unable to analyze ill-formed documents, many systems insist that the user correct any newly introduced "errors" before proceeding. This treatment has unpleasant side effects.

- It narrows options available to the user, who may prefer to delay trivial repairs while dealing with more important issues. The "error" may be part of an elaborate textual transformation.
- It implies that derived information is only available and accurate when documents are well-formed, again constraining the user.
- It implies that the user has done something wrong, when in fact the system is simply unable to understand what the user is doing [LN86].

I2: Incompleteness. This is an important special case of ill-formedness; it corresponds to natural intermediate states for documents under construction. Language-based information for software that is generally well-formed but incomplete should be first class. Some systems address this with "placeholders," visible glyphs that appear in places corresponding to unexpanded nonterminals in the internal derivation tree, but more flexible versions of this are needed.

I3: Inconsistency. Any situation where one kind of information (syntax tree, for example) is derived from another (text) invites inconsistency between the two when things change.[1] This presents a dilemma for language-based editing systems. Derived information (including diagnostics concerning ill-formedness) produced by analyzing text is only trustworthy immediately after an analysis. On the other hand, not to support any language-based services in this state is a needless interruption of service, since most of that information is correct most of the time.

3.6 Description-driven support for multiple languages

Software engineers use many languages: design languages, specification languages, structured-documentation languages, programming languages, small languages for scripts, schemas, mail messages, and embedded "little languages" [Ben86]. Bernard Lang commented that "Language independence is essential for the adaptability of the environment to different dialects or to the evolution of a language. It is also a factor of uniformity between environments for different languages" [Lan86].

A language-based editing system must support multiple languages smoothly and uniformly, even permitting switching among languages during single working sessions. It must be as convenient as possible to add support for new languages using natural, declarative, language-description mechanisms, which allow a description writer to focus on what is being described rather than on how document analyzers operate.

3.7 Extensibility and customizability

An effective language-based editing system must be customizable and extensible in order to accommodate the enormous variations among individual users, among projects (group behavior), and among sites [Lan86][Sta81].

A language-based editing system must be capable of using a variety of information, derived by many different tools in the environment. Users opportunistically exploit many forms of information to help them understand and modify complex documents [Let86]. "Information gathering" is the primary task associated with important activities such as program maintenance [HBS87].

4 The *Pan* Prototype

Pan I version 4.0 [DV91] is a fully implemented, multilingual, language-based editing and browsing system that addresses the usability requirements presented in Section 3. This prototype 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 [GHM92].

Figure 2 shows two *Pan* views of a simple program. The larger view displays editable program text, typeset and marked up with colored highlighters that reveal instances of

1. Inconsistency should not be confused with ill-formedness. Inconsistency means that the system cannot determine whether a document is well-formed or not.

Figure 2: Two *Pan* views on a simple program

"Language Error," where "Language Error" is the category of meta-information under investigation at the moment by the user. The smaller view displays a projection of the same category into an otherwise generic list-oriented view that shares structural navigation with the larger view. The following two sections describe some of the design solutions embodied by this prototype.

5 Design Metaphors

The requirements of Section 3 demand a system rich in functionality. Making such a system effective for software engineers presents user-centered [Nor86] design problems:

- How to deliver services without overwhelming users with complexity unrelated to their tasks, and
- How to enable users experienced with conventional text editors to transition productively.

A useful strategy is to articulate *design metaphors* that summarize how users are expected to understand the resulting system. For example, every aspect of *Pan*'s design, implementation, and configuration supports the following five metaphors, with the explicit goal that users will understand them without being told.

5.1 Augmented Text Editor

The system is a text editor whose text-based services are always available in every context. All other services increase user options: some may be used to guide text editing but they never interfere with it. A useful analogy for the other services would be spelling checkers in document processing systems.

5.2 Heads-Up Display

Many services show information *about* documents as enhancements to the text display. This approach is analogous to "heads-up" display in which data are displayed by overlay onto pilots' primary visual field, allowing them to attend continuously to the most important part of their job: looking and flying. The primary visual field here is the mostly textual display of software documents from which the user should be distracted as little as possible.

5.3 Imperfect World

Although the system exploits knowledge of underlying languages, it operates no more differently in the presence of "language errors" than does a text editor in the presence of spelling errors.

5.4 Smart versus Dumb Services

Many language-based services appear to users not as distinct mechanisms (along with the confusion of extended command sets), but as optional generalization of familiar, text-based services. A generalized service typically changes character dynamically: *dumb* when operating textually, *smart* when operating with the additional advantage of language-based information. Unobtrusive visual cues reveal whether a particular service is smart or dumb at any moment.

5.5 Strict versus Gracious Services

Many language-based services can operate during periods of inconsistency, when language-based information derived from text is out of date and therefore unreliable. These gracious services are characterized by shifts between *exact* and *approximate* modes of operation, with little apparent change in behavior, but with unobtrusive visual cues that reveal the current mode. Strict services, on the other hand, operate not at all during periods of inconsistency: a strict service may simply become dumb when a document becomes inconsistent, or it may trigger analysis in order to proceed.

6 Architectural Solutions

The implementation of a system both flexible and powerful enough to support the requirements of Section 3 and the design metaphors of Section 5 requires careful separation of concerns. For example, *Pan*'s design framework exploits the architectural layering shown in Figure 3 and permits reuse of many components for construction of additional services.

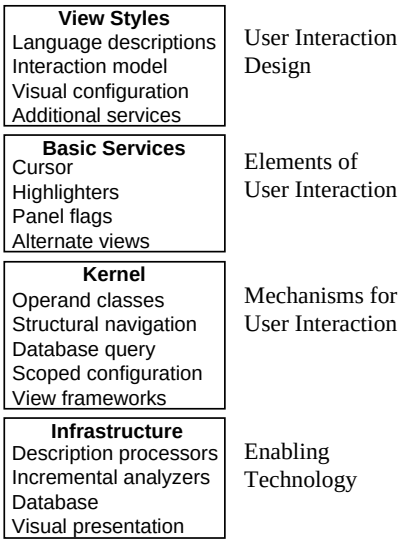

Figure 3. Architectural Design Layers

6.1 Infrastructure: Isolation of Language-Based Technology

It is tempting think of language-based editing systems as interactive compilers, but technology developed for compilers ports badly into the domain of user interaction. The lowest layer of this architecture isolates language-based analysis mechanisms; in *Pan*, for example, this layer includes the *Ladle* [But89] and *Colander* [Bal89] subsystems for description-driven generation of incremental syntactic and static-semantic analyzers respectively. Among the goals for these two subsystems is the useful representation of ill-formed programs.

The rest of the system accommodates flexible, user-centered design choices without excessive coupling to the batch-oriented, compiler model of software structure. An experimental static-semantic analyzer was substituted for *Colander* at one point without serious difficulty.

6.2 Kernel: Basic Abstractions

The second layer of this architecture implements essential abstractions upon which user interaction rests. These abstractions are designed to be language-independent and flexible.

For example *Pan*'s operand class mechanism permits convenient definition (by view

style designers, see below) of arbitrary sets of structural elements that are computed dynamically as the representation changes. An operand class definition may be defined intentionally as a predicate on nodes of the internal tree, drawing on any information in the database, or extensionally by reference to some outside agent that enumerates membership as needed. An operand class has a "title" for communication with users, for example "Statement," "Language Error," "Integer Expression," or "Disallowed Coercion." Together with the services it can be configured to drive, an operand class dynamically creates a new concept in the user's model of interaction with structure of software. The operand class solves several problems in user interface design and permits services to be adapted for uniform operation across multiple language-based view styles.

6.3 Basic Services: Reusable Elements

Services that the user sees are constructed in the third layer of this architecture, designed to be as simple and flexible as possible. Most of *Pan*'s Basic Services are designed to be configured by operand class definitions, for example as shown in Figure 2:

- A *Panel Flag* with the appearance of an exclamation point is configured to appear in the upper right hand corner of the window whenever any instances of operand class "Language Error" are present.
- A *Highlighter* is configured to display with red ink all text corresponding to instances of "Language Error."
- *Structural Navigation* moves the cursor to the next instance of the current "Level," which the user may select from a panel menu or by keystroke accelerators. In the example the cursor has just moved to an instance of class "Language Error." The structure cursor is on the text "N 1" as a result, but it might have landed at the same place had the user requested a move to the next "Expression." A structural element can be in many classes simultaneously.
- *Alternative Views* present information organized in different ways, for example as does the view named "[Language Error]" in Figure 2. The generic "list view" service is configured to display (or "project") instances of "Language Error," and to display a diagnostic in place of the usual text. This view supports shared navigation: selection of a diagnostic from the list causes coordinated navigation in all other views.

As suggested by this example, the notion of "error" does not exist in this architecture at either the Kernel or Basic Services layers; it is merely one of many kinds of information available in the Infrastructure. Errors, like other concepts in the user's model of software structure, are managed in this layer by configuration.

6.4 View Styles: Configuration by Design

Adaptation to working contexts is captured in the fourth layer of this architecture by the notion of multiple *view styles* for user interaction, each specialized for a particular combination of user population, language being used, and task at hand. A view style:

- includes traditional syntax and static-semantic language descriptions, but may extend to extra-lingual analysis such as stylistic and usage guidelines;

- specifies services to be provided and specializes generic services;
- defines a visual context, including typography and use of color; and
- configures details of interaction, including keystroke and menu-bindings.

A human "view style designer" necessarily creates view styles, from which much of the richness and effectiveness of this framework derive. Part of the view style designer's task is to define a conceptual vocabulary (in the form of operand classes) describing program structure that is appropriate to the language being used, to the intended user population (different kinds of users will want different view styles, even for the same language), and to different tasks (some users may want view styles specialized for particular tasks, design recovery vs. exploratory programming for example).

A working system includes a suite of view styles that collectively offers appropriate services and uniform user interaction across all styles. This framework cannot guarantee good design by view style designers, but it provides tools, guidelines, and examples, among which are solutions to usability problems that plague earlier generations of systems.

6.5 Applications

Finally, this architecture is *open*. To realize the full power of language-based interaction, the editor must function as an interface through which many language-related services can be delivered to software engineers. Known as *applications* in this framework, these additional services can be added using an extension language, configuration mechanisms for reuse of Basic Services, and an extensible data repository. They can also be delivered by integration with other tools, for example allowing the system to serve as a user interface for compilers, profilers, debuggers, and code auditors.

A number of these applications have been prototyped, none of which require specialized support in the Infrastructure layer . All are constructed using the Kernel and Basic Abstraction layers; they are language-independent and use familiar interaction paradigms for uniformity.

- Semantically sensitive variable renaming.
- Semantics-based query, for example highlighting all uses of a name or showing its type.
- Semantics-based navigation, for example jumping to the declaration of a name.
- Variable cross reference view.
- Module table of contents view.
- Syntax-directed editing, as supported by many structure editors. Figure 4 shows *Pan* as a user is about to expand a placeholder in a toy language.
- Style checking, for example type-sensitive naming conventions.
- Textual annotations on structural elements.
- Debugger integration [BFG92].

Many more applications would be straightforward to implement within this architec-

Figure 4: A syntax-directed editing application in *Pan*

ture:

- Operand classes for members of particular libraries, with a slight extension providing hyper-links to documentation.
- Extended type checking for library calls that take complex argument sequences, for example “printf” in C and window system libraries such as Xview [Hel90].
- Operand classes for software reengineering, for example uses of “goto” or non-portable types, along with appropriate summary views.
- Operand classes for information imported from tools in the environment, for example profile information, dead code analysis, and test coverage.

- An operand class for recently created structure, along with appropriate highlighter.

The following, more speculative applications would exploit this basic design framework even further, but would require additional mechanisms and further integration with other tools.

- Graphical cues, for example small glyphs as suggested by Baecker and Marcus for categories such as “Warning/Sensitive,” “Fragile Code,” and “Unreachable Code” [BM90].
- Redundant notation to aid comprehension, for example control scope markings as suggested for certain maintenance tasks [SGG77].
- Transformed notation to aid comprehension, for example alternative representations of deeply nested conditionals [PND87], especially when placed *in situ* in the display so that continuity with surrounding context is not disturbed.
- Displayed program “slices” [Wei84].
- Specialized debugger interface, for example a multiple view debugger designed for optimized code [BHS92].

7 Conclusions and Open Issues

Simplicity and Usability: Naive solutions to the requirements set forth in Section 3 would drown users in a sea of system-induced complexity. The design framework developed as part of this research, applied with a judicious mix of attention to users and tasks, carefully chosen design metaphors, and flexible abstractions, demonstrates that crucial simplicity and usability can be achieved.

Integration with Other Tools. The services provided by a language-based editing system alone cannot justify the cost of the technology. This architecture addresses the useful viewing of a wide variety of possibly large-scale information, but the tremendous leverage this offers can 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.

Language-Based Technology. New language technology was developed during the project specifically to support user interaction [BBG88][Bal89][But89]. Nevertheless limitations caused by its batch-oriented compiler heritage still managed to appear. The apparent boundary between editing and compiling should become more blurred (as it will between editing systems, compilers, and their underlying languages). This demands aggressive factorization into components, with each component of the technology being further developed for this new, more general role.

Advanced Visual Presentation. The advantages of high-quality visual presentation are clear, but some techniques based on the static book publishing paradigm were not supported by the prototype rendering engine. The reasons are instructive:

- Just as batch-oriented compiler technology does not support interaction well, some design choices made for static publishing are not appropriate in a more dynamic context.
- Limited display screen resolution condemns some techniques to ineffectiveness or outright invisibility.
- Implementation was costly, and four generations of toolkits offered no useful support for the kind of visual information layering demanded by the application.

User Experience. Although informal user feedback has guided much of this work, more empirical evidence is needed. This kind of experience can only be gathered by experimenting with a flexible system such as *Pan* in production environments using production languages.

Annotation: An important part of the software engineer's task is the creation and reading of annotations, typically in the form of textual comments. *Pan* delivers no support for this task beyond that provided by an ordinary text editor.

Object-Oriented Programming. Much of the experience and insight that drove this research 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 these techniques will apply, but new ones may be needed to accommodate changing notions of system modularity and connectivity.

Language Extension. The static language description and analysis model adopted for the project 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. User interaction techniques described here should apply in most cases, but they may need to be adapted (as the language analysis model must change) for these more dynamic contexts.

8 Acknowledgments

Special thanks go to Robert Ballance, Jacob Butcher, and Prof. Susan L. Graham, from whose collaboration many of *Pan*'s good ideas emerged and were realized. Important contributions also came from Christina Black, Laura Downs, Bruce Forstall, Mark Hastings, Darrin Lane, and William Maddox. Yuval Peduel and Jiho Sargent provided many constructive comments on an earlier draft of this paper.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DoD), monitored by the Space and Naval Warfare Systems Command under Contract N00039-88-C-0292, by the Defense Advanced Research Projects Agency under Grant MDA972-92-J-1028, by IBM under IBM Research Contract No. 564516, by a gift from Apple Computer, Inc., a State of California MICRO Fellowship, and an IBM Fellowship.

9 References

[BM90] Ronald M. Baecker and Aaron Marcus, Human Factors and Typography for More Readable Programs, Addison-Wesley Publishing Co. (ACM Press), Reading, MA, 1990.

[BS86] 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.

[BS92] Rolf Bahlke and Gregor Snelting, Design and Structure of a Semantics-Based Programming Environment, *International Journal of Man-Machine Studie*s 37,4 (October 1992), 467-479.

[BBG88] Robert A. Ballance, J. Butcher and Susan L. Graham, Grammatical Abstraction and Incremental Syntax Analysis in a Language-Based Editor, Proceedings of the ACM-SIGPLAN 1988 Conference on Programming Language Design and Implementation 23,7 (June 22-24, 1988), 185-198.

[Bal89] Robert A. Ballance, "Syntactic and Semantic Checking in Language-Based Editing Systems", UCB/CSD-89-548, Ph.D. Dissertation, Computer Science Division, EECS, University of California, Berkeley, December 1989.

[BGV92] 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.

[Ben86] Jon Bentley, Little Languages, *Communications of the ACM* 29,8 (August 1986), 711-721.

[BCD+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.

[BFG92] John Boyland, Charles Farnum and Susan L. Graham, "Attributed Transformational Code Generation for Dynamic Compilers", in *Code Generation -- Concepts, Tools, Techniques*, Robert Giegerich and Susan L. Graham (editors), Springer Verlag, Berlin, 1992

[BHS92] Gary Brooks, Gilbert J. Hansen and Steve Simmons, A New Approach to Debugging Optimized Code, *Proceedings of the ACM-SIGPLAN 1992 Conference on Programming Language Design and Implementation* 27,7 (June 17-19, 1992), 1-21

[But89] Jacob Butcher, "Ladle", UCB-CSD-89-519, Computer Science Division, EECS, University of California, Berkeley, November 1989. Master's Thesis.

[CMP+90] Ravinder Chandhok, Phillip Miller, John Pane and Glenn Meter, "Structure Editing: Evolution Towards Appropriate Use", Presented at the CHI '90 Workshop on Structure Editors, Seattle, Washington, April 1990.

[CMP91] D. D. Cowan, E. W. Mackie and G. M. Pianosi, Rita--an editor and user interface for manipulating structured documents, *Electronic Publishing* 4,3 (September 1991), 125-150

[DV91] Laura M. Downs and Michael L. Van De Vanter, "*Pan* I Version 4.0: An Intro-

duction for Users", 91/659, Computer Science Division, EECS, University of California, Berkeley, August 1991.

[DHK+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.

[Gol87] Adele Goldberg, Programmer as Reader, *IEEE Software* 4,5 (September 1987), 62-70.

[GL88] Dennis R. Goldenson and Marjorie B. Lewis, Fine Tuning Selection Semantics in a Structure Editor Based Programming Environment: Some Experimental Results, A*CM SIGCHI Bulletin* 20 (October 1988).

[GHM92] 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.

[Hel90] Dan Heller, *XView Programming Manual: An OPEN LOOK Toolkit for X11*, O'Reilly & Associates, Inc., Sebastopol, California, 1990.

[HBS87] Robert W. Holt, Deborah A. Boehm-Davis and Alan C. Schultz, "Mental Representations of Programs for Student and Professional Programmers", in *Empirical Studies of Programmers: Second Workshop*, Gary M. Olson, Sylvia Sheppard and Elliot Soloway (editors), Ablex Publishing, Norwood, New Jersey, 1987, 33-46.

[Joh85] Vera John-Steiner, *Notebooks of the Mind: Explorations of Thinkin*g, Harper & Row, 1985.

[KR91] Jurgen Koenemann and Scott P. Robertson, "Expert Problem Solving Strategies for Program Comprehension", Proceedings SIGCHI Conference on Human Factors in Computing Systems, New Orleans, Louisiana, 1991, 125-130.

[Lan86] Bernard Lang, "On the Usefulness of Syntax Directed Editors", in Advanced Programming Environments, Lecture Notes in Computer Science vol. 244, Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), Springer Verlag, Berlin, 1986, 47-51

[Ler92] Barbara Staudt Lerner, Automated Customization of Structure Editors, *International Journal of Man-Machine Studies* 37,4 (October 1992), 529-563

[Let86] Stanley Letovsky, "Cognitive Processes in Program Comprehension", in *Empirical Studies of Programmers*, Elliot Soloway and Sitharama Iyengar (editors), Ablex Publishing, Norwood, New Jersey, 1986, 58-79.

[LS86] Stanley Letovsky and Elliot Soloway, Delocalized Plans and Program Comprehension, *IEEE Software* 3,3 (May 1986), 41-49.

[LN86] Clayton Lewis and Donald A. Norman, "Designing for Error", in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper (editors), Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 411-432.

[Min90] Sten Minör, On Structure-Oriented Editing, PhD Dissertation, Department of Computer Science, Lund University, Sweden, January 1990.

[Nea87] Lisa Rubin Neal, “Cognition-Sensitive Design and User Modeling for Syntax-Directed Editors”, Proceedings SIGCHI Conference on Human Factors in Computing Systems, Toronto, Canada, April 1987, 99-102.

[Nor86] 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.

[Not85] David Notkin, The GANDALF Project, *Journal of Systems and Software* 5,2 (May 1985), 91-105.

[OC90] Paul Oman and Curtis R. Cook, Typographic Style is More than Cosmetic, *Communications of the ACM* 33,5 (May 1990), 506-520.

[PND87] G. R. Perkins, R. W. Norman and S. Danicic, Coping with Deeply Nested Control Structures, *SIGPLAN Notices* 22,2 (February 1987), 68-77.

[RT84] 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.

[RT87] Thomas Reps and Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Department of Computer Science, Cornell University, 1987. Second edition.

[RW88] Charles Rich and Richard C. Waters, The Programmer's Apprentice: A Research Overview, *Computer* 21,11 (November 1988), 11-25.

[Sch83] Donald A. Schoen, *The reflective practitioner: how professionals think in action*, Basic Books, New York, 1983

[SGG77] M. E. Sime, T. R. G. Green and D. J. Guest, Scope Marking in Computer Conditionals -- A Psychological Evaluation, *International Journal of Man-Machine Studies* 9 (1977), 107-118.

[SE84] Elliot Soloway and Kate Ehrlich, Empirical Studies of Programming Knowledge, *IEEE Transactions on Software Engineering* SE-10,5 (September 1984), 595-609.

[Sta81] Richard M. Stallman, EMACS: The Extensible, Customizable, Self-Documenting Display Editor, Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, *SIGPLAN Notices* 16,6 (June 8-10 1981), 147-156.

[TR81] Tim Teitelbaum and Thomas Reps, The Cornell Program Synthesizer: A Syntax-Directed Programming Environment, *Communications of the ACM* 24,9 (September 1981), 563-573.

[Van92] 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.

[VGB92] 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.

[Wei84] Mark Weiser, Program Slicing, *IEEE Transactions on Software Engineering* SE-10,4 (July 1984), 352-357.

[Win79] Terry Winograd, Beyond Programming Languages, *Communications of the ACM* 22,7 (July 1979), 391-401