Displaying and Editing Source Code in Software Engineering Environments

Michael L. Van De Vanter[1] and Marat Boshernitsan[2]

[1]*Sun Microsystems Laboratories*
901 San Antonio Avenue
Palo Alto, CA 94303 USA
Tel +1 650 336-1392, Fax +1 650 969-7269, Email michael.vandevanter@sun.com

[2]*Department of Computer Science*
University of California at Berkeley
Berkeley, CA 94720-1776 USA
Tel +1 510 642-4611, Fax +1 510 642-3962, Email maratb@cs.berkeley.edu

Abstract

Source code plays a major role in most software engineering environments. The interface of choice between source code and human users is a tool that displays source code textually and possibly permits its modification. Specializing this tool for the source code's language promises enhanced services for programmers as well as better integration with other tools. However, these two goals, user services and tool integration, present conflicting design constraints that have previously prevented specialization. A new architecture, based on a lexical representation of source code, represents a compromise that satisfies constraints on both sides. A prototype implementation demonstrates that the technology can be implemented using current graphical toolkits, can be made highly configurable using current language analysis tools, and that it can be encapsulated in a manner consistent with reuse in many software engineering contexts.

Keywords: Program editor, software engineering tool integration, language-based editing

1. Introduction

Any interactive software engineering tool that deals with programs inevitably displays source code for a human to read and possibly modify[1]. The technology for doing this, however, has changed little in twenty years, despite a compelling intuition that specializing the technology for programming languages might increase user productivity substantially. In contrast, consider how word processing systems have evolved beyond simple text editors during those same twenty years.

Extensive research, numerous prototypes, and more than a few commercial attempts have failed to deliver practical language-based editing for source code. Programmers find such systems difficult and unpleasant when compared with simple text editors. Tool builders find that implementations are fragile and place high demands on supporting infrastructure.

Language-based editing will only succeed in practice when it addresses the real goal: to help programmers program in the context of existing skills and tools. This translates to two sets of requirements, often conflicting, for an editor:

- *Programmer's perspective*: the editor must make reading and writing source code easier and more rewarding.
- *Tool builder's perspective*: the editor must reliably share information with other tools, for which it may act as a user interface, and it must be packaged for reuse (portable, highly configurable, and embeddable).

The CodeProcessor[2] is an experimental tool for editing source code, under development at Sun Microsystems Laboratories. It is based on technology that strikes a balance among apparently competing requirements. It is text oriented, but fundamentally driven by language technology. It can make its language-oriented representation (configured by declarative specifications) available to other tools, and can be embedded in other GUI contexts. The key architectural choice is a lexically-oriented intermediate representation for source code that addresses both usability and integration with other tools.

---

1. We do not address purely graphical programming languages, although some of the issues are similar.

2. "CodeProcessor" is an internal code name for this prototype; it is intended to suggest a specialization of simple text editing for source code, much as word- and document-processors are specialized for natural language documents.

Experience suggests that simple usability testing, better GUI design, or new algorithms would not have produced this design. Rather, it resulted from rethinking the tasks, skills, and expectations of programmers, and from then finding ways to address them: using existing language technology and within the context of practical software engineering tools. The result is an architecture that is different, though not necessarily more complex, than those tried in the past.

This paper presents an overview of the CodeProcessor and the design choices it embodies. Section 2 reviews requirements, and Section 3 discusses how previous technologies have failed to meet them all. Section 4 offers a new look at the design trade-offs needed when combining text editing and language support, and shows how this analysis leads to a solution. Sections 5 and 6 describe the two complementary and mutually dependent aspects of the CodeProcessor's design: architecture and user-model. Finally Section 7 reviews implementation status, followed by related work and conclusions.

2. Design goals

The requirements mentioned in the introduction, and discussed in more detail here, reflect different perspectives: programmers and tool designers. Past failures result from neglecting one point of view or the other; Sections 3 and 4 will show how they can be reconciled.

2.1. No training

All available evidence shows that programmers read programs textually; they also have "structural" understanding, but it is highly variable and not based on language analysis [10][12]. Programmers have deeply ingrained work habits as well as motor-learning that involves textual editing; they will only accept a tool that is familiar enough for immediate and comfortable use without special training.

This need not, however, prohibit advanced functionality. Consider how users experienced with simple text editors find the transition to word processors smoothed by familiar text entry and cursor commands. By analogy, language-based editing services should be layered carefully onto basic text editing behavior, imposing no (or barely noticeable) restrictions.

2.2. Enhance reading and writing

Additional editing services derive from specialization for the tasks confronting programmers. A familiar example is automatic indentation of source code lines. This service is based loosely on linguistic structure, and it helps both reading (visual feedback on nesting) and writing (saving tedious keystrokes). This particular service can be delivered in a simple text editor, but it can and should be taken much further.

Research shows that high quality, linguistically-driven typography measurably improves reading comprehension [3][19]. In many environments, reading is still the dominant task for programmers, even while writing code [9][31]. Good designs for program typography are available (for example the paper-based publication designs by Baecker and Marcus [3]), yet rarely used.

Also highly important, is special support (both reading and writing) for program comments. Transparent to conventional language tools, comments are tedious to format but crucial for readers.

Although specialized enhancements are important, it is absolutely essential that they not make things worse. Any intrusion on text editing must respect the "balance of power" between user and tool. This can be delicate even in the simplest of cases, for example auto-indentation mechanisms that programmers find helpful but "not quite right."

Nowhere has intrusiveness been more problematic than in treatment of fragmentary and malformed source code. This is, of course, the normal state for programs under development. Unfortunately, language-based editors typically treat such situations as user "errors" and encourage or require corrective action. The real "error" is that the tools fail to model what the user is really doing [14] and cannot function usefully until rescued. Editing tools must function without interruption in any context.

2.3. Access to linguistic structure

Software engineering tools (for example analyzers, builders, compilers, and debuggers) generally operate over structural source code representations such as abstract syntax trees. An editing tool is most easily integrated with other tools if it can share such representations, but as Section 3.1 discusses, this presents severe design challenges for a tool whose job is to display and permit modification to source code in terms of text.

2.4. Configuration and embedding

Finally, as software engineering tools evolve, emphasis shifts from standalone editing systems to specialized tools that must work with other tools. A tool for source code editing must be well encapsulated, somewhat like a GUI component, and not demand complex support such as a particular kind of source code repository. Reflecting the reality that practical software engineering involves many languages, it should be easily configured via language specifications. In order to be used as an interface by many other tools, an editing tool must have a visual style that is easily configured for different contexts and tasks.

3. The design space

At the heart of a specialized editing tool is an internal representation for source code. Conventional choices, depicted in Figure 1, are divided by a gulf between fundamentally different approaches: one oriented toward usability and one toward higher level services.

Text   Text
\+
Ad Hoc Matching

Language Structure   Language
\+   Structure
Text Escapes

Simple user-model
(less structural: better for users)

Rich representation
(more structural: better for services)

Figure 1: Design choices for program editors

3.1. Pure designs

At the far right of the diagram are "structure editors" [4][6][8][18], so called because of internal representations closely related to the tree and graph structures used by compilers and other tools. This greatly simplifies some kinds of language-oriented services, but it requires that programmers edit via structural rather than textual commands. Behind this approach is a conjecture, articulated by Teitelbaum and Reps, that programs are intrinsically tree structured, and that programmers understand and should manipulate them that way [25]. Unfortunately, years of failed attempts [11], combined with research on program editing [17] and on how programmers really think about programs [13][22] have refuted that conjecture. From a tool integration perspective, the advantages of complete linguistic analysis are offset by its fragility (in the presence of user editing) and context-dependency (the meaning of code in many languages depends potentially on all the other code with which it will run). Few structure editors are in use today

At the far left are simple text editors with no linguistic support. Editing is simple and familiar, but there is no real specialization for source code. Integrating a simple text editor with software engineering tools requires complex mappings between structure and text, but these typically result in restrictive and confusing functionality, fragile representations (for example, where the identity of structural elements is not preserved during editing operations), or both [27].

3.2. Modified designs

Subsequent efforts in language-based editing can be viewed as attempts to bridge this gulf. Some structure editors allow programmers to "escape" the structure by transforming selected tree regions into plain text [21], but usability problems persist. The complex unseen relationship between textual display and internal representation makes editing operations, both structural and text escapes, confusing and apparently unpredictable [27] because of "hidden state." Textual escapes make matters with a confusing and distracting distinction between those parts of the program where language-based services are provided and those where they are not. Often language services and tools stop working until all textual regions are syntacti-

cally correct and promoted back into structure.

At the left side of Figure 1 are widely used code-oriented text editors such as Emacs [23]. These use a purely textual representation, assisted by ad-hoc regular expression matching that recognizes certain language constructs. The structural information computed by simple text editors is, by definition, incomplete and imprecise. It therefore cannot support services that require true linguistic analysis, advanced program typography for example. Simple text editors typically provide indentation, syntax highlighting[1] and navigational services that can tolerate structural inaccuracy. A malformed program will, at worst, be incorrectly highlighted.

A few text editors perform per-line lexical analysis with each keystroke, but the information has never been fully exploited and the lack of a true program representation leads to confusion in the inevitable presence of mismatched string quotes and comment delimiters.

3.3. Inclusive designs

A more inclusive approach is to maintain both textual and structural representations. Although this approach promises a number of advantages [5][26], it is difficult to keep the representations consistent and it has not been demonstrated that the cost and complexity are justified.

4. Finding the middle ground

Section 3 described a fundamental design tension:
- It is desirable to maintain a linguistically accurate program representation, updating it on every modification, however small.
- The greater the degree of structural sophistication, the more fragile the representation is in the presence of unrestricted textual editing, and the more room there is for confusing behavior and inconsistency between what's seen and what's represented internally.

In summary, an ideal representation would be closely related to displayed text, but would also reflect linguistic structure at all times. What's needed is a compromise

---

1. "Syntax highlighting" is an unfortunate misnomer, since pattern-matching is considerably weaker than syntactic analysis. It would be more accurate to call it "unreliable keyword, string, and comment recognition".

| Pure Text | Text + Regular Expressions | Lexical Tokens | Lexical Tokens + Fuzzy Parsing | Parse Tree | Syntax Tree + Fuzzy Attribution | Fully-Attributed Syntax Tree |

Simple user-model
(less structural: better for users)

Rich representation
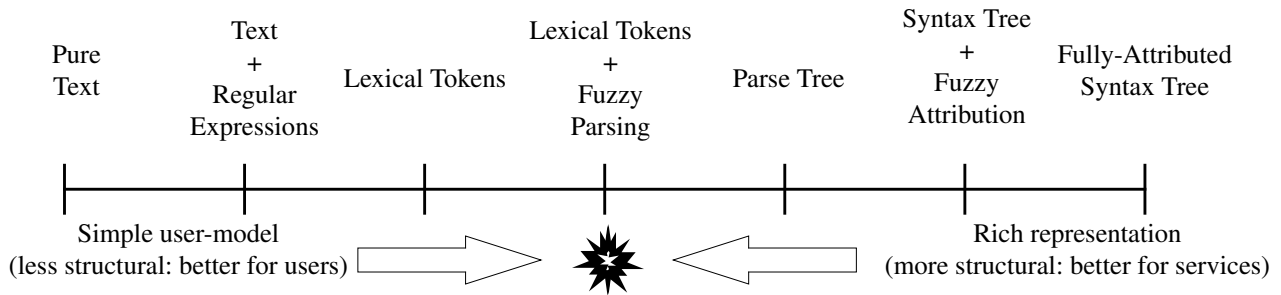(more structural: better for services)

Figure 2: Additional choices for program representation and analysis

somewhere in the middle of Figure 1, where the amount of language analysis performed is as simple (and localized) as possible, but also as useful as possible.

A compromise can be found by taking a closer look at language analysis: both the internal engineering of compilers, and the formal language theory behind it. A typical compiler analyzes textual programs in phases, shown below. Each stage is driven by a different kind of grammar

lexical analysis → parsing → static semantic analysis

(corresponding approximately to types 3, 2, and 1 in the Chomsky grammar hierarchy) and uses a corresponding kind of analyzer [29]. Programming languages are often designed around this grammatical decomposition, and batch-oriented compilers benefit from the simplicity and formal foundations of separate phases.

This decomposition reveals additional choices, depicted in Figure 2, for analyzing and representing programs being edited. Possible representations include the standard products of each phase: lexical token stream, parse tree, and attributed tree respectively. Intermediate choices include partial analysis of the next grammatical level: regular expression matching is a partial lexical analysis, fuzzy parsing is a partial syntactic analysis which recognizes only certain features of the context-free syntax (e.g. nested parenthesis or context-dependent categorization of identifiers into function and variable names), and partial semantic attribution that can be used for computing limited amounts of semantic context. Partial analyses are often simpler to implement (fuzzy parsing can be performed through a simple pattern matching on the token stream) and more forgiving of inconsistencies in the representation.

An important distinction among the three analysis phases concerns the scope of cause and effect. Static semantic analysis (closely related to Chomsky's context-sensitive syntax) at each point in a program depends potentially upon the entire program. Parsing (context-free syntax) depends only on the enclosing phrase, but assumes that program is well formed. Lexical analysis (regular syntax) depends only on adjacent tokens, making it highly suitable for the inner loop of an editor.

Thus the lexical representation, not used in any prior systems, emerges as a promising compromise:

- It is a stream, not a tree, and thus bears a close relationship to textual source code;
- The analysis needed to update the representation after each edit usually requires only local context;
- It is suitable for program fragments;
- It has enough linguistic information to provide many language-based services, including more robust implementation of familiar services such as indentation, parenthesis and bracket matching, procedure or method head recognition, etc.; and
- It is a language representation suitable for integration with other tools, including complete language analyzers. Further analysis, for example parsing, could be folded into the CodeProcessor if added carefully, but at some additional cost in complexity.

Although this approach is promising, a number of design questions remain:

- Can the textual display and behavior be made to look and feel familiar enough that it requires no training?
- To what degree can the display be specialized for programs using only lexical information?
- Can such a fine-grained typographical display be implemented using current toolkit technology and made configurable?
- Can the lexical token representation be made robust in the presence of partially typed and badly formed tokens? In particular, how can "bracketed" tokens such as string literals be managed when one of the brackets (double quotes for strings) is missing?
- What specialized support for comments and other, possibly non-textual, annotations is possible?
- How can a description-driven lexical analyzer be adapted to update the representation after each keystroke?

Solutions appear in the following two sections, which summarize respectively the two mutually dependent aspect of the CodeProcessor's design: architecture/implementation and user-model. The architecture is presented first in Section 5, although many aspects were driven by the user-model design described in Section 6.
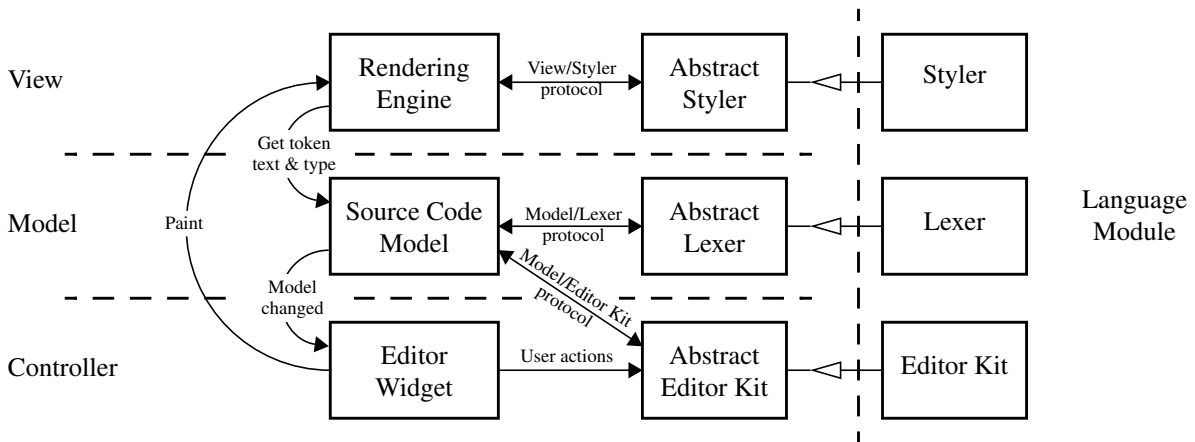
Figure 3: CodeProcessor Architecture

5. Architecture

The CodeProcessor's architecture, depicted in Figure 3, is based on the Model-View-Controller design paradigm. This choice is not accidental: in addition to being a natural architecture for display and editing, it also reflects the design of the Java™ Foundation Classes (JFC) "Swing" toolkit and its text framework [30] which was used to implement the current prototype. Multi-lingual behavior is supported by separating each of the three core modules into two components: one implementing the language-independent functionality, and the other (collectively known as a *Language Module*) providing language-sensitive features for a particular language. In the CodeProcessor this separation is achieved by subclassing, but other decompositions are also possible.

The remainder of this section describes each of the major design constituents in order.

5.1. The Controller

The *Controller* is manifested through two closely related components: the *Editor Widget* and the *Editor Kit*. The Editor Widget is responsible for dispatching window system events and making the CodeProcessor a fully functional member of the JFC widget family. The Editor Kit implements the intricate editing behavior described in Section 6.2.

Much of Editor Kit's functionality is language-independent; some, however, may be custom-tuned for each particular language, for example adding keyboard shortcuts for inserting language constructs.

The primary responsibility of the Editor Kit is to implement user actions that require taking the context of the action into the consideration. Some actions, such as cursor movement commands, require no changes to the source code model; their execution depends only on the context (tokens) surrounding the cursor. Other actions, such as insertions and deletions, may depend not only on the modification context, but also on the state *after* the modification, since certain nuances of the user-model require "looking into the future."

To facilitate this, the Editor Kit commences a two-stage modification process upon any potential change. First, the source code model is requested to consider the effects of the change *without* modifying the underlying content. This produces an object describing the change in terms of a model transformation that needs to take place. When the Editor Kit regains control it examines the transformation, either discarding it, if it has no effect or is not valid, or applying it to the model.

5.2. The Model

As discussed in Section 4, source code is represented as a sequence of lexical tokens, although this representation is extended in several crucial ways. This representation allows for much-needed flexibility, as it both supports the required user-model, and fits naturally with the incremental lexical analysis algorithm.

The lexical analysis algorithm, developed by Tim Wagner [28], is fully general: it supports unbounded contextual dependency and multiple lexical states. Moreover, incrementality can be crafted onto existing batch lexers that conform to a simple interface. For instance, the current prototype's lexer for the Java programming language is generated by the JavaCC tool [16] from a readily available lexical specification; the specification is extended to include various categories of irregular lexemes created during editing, as discussed in Section 6.1.

Figure 4 depicts the modification of a model after insertion of the characters "=x" into a fragment containing the four tokens 'a', '+', 'c', and ';' with cursor initially between '+' and 'c'. Figure 4a represents the

content immediately prior to the modification, 4b -- the transformation resulting from considering given modification, and 4c -- the content after the suggested transformation has been applied.
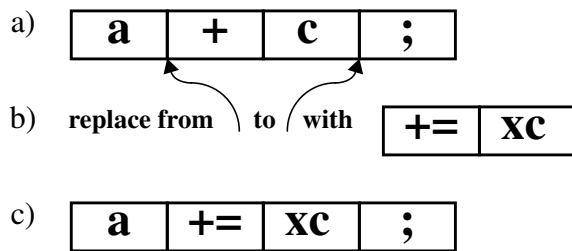

a)

| **a** | **+** | **c** | **;** |

b)   **replace from** \ **to** / **with** | **+=** | **xc** |

c) | **a** | **+=** | **xc** | **;** |

Figure 4: Example model update

The source code model is also responsible for adding and removing "separators," special non-linguistic tokens whose role in the user-model is described in Section 6.2. Other non-linguistic tokens include comments, line breaks, and other layout directives.

A significant advantage of the model, from the perspective tool integration is that it enables *stable* references to source code structure: during any kind of editing, the *identity* of unaffected tokens is guaranteed.

5.3. The View

The rendering mechanism displays source code in accordance with the requirements outlined in Section 6.1. The typographically-enhanced display is facilitated by assigning stylistic properties to each token by means of the *Styler* component. The Styler lends itself to being automatically-generated, although the current implementation uses hand-written Stylers.

Stylers can also be used to export human-readable source code from the CodeProcessor by rendering into a character stream, dropping stylistic information that cannot be represented. Appropriate formatting can be achieved by Stylers optimized for text output.

5.4. Representing embedded structures

Programming languages commonly include embedded syntactic structures that have distinct lexical rules, most notably comments and strings. Embedded structures are supported by nested editors with transparent boundaries (behavioral considerations are presented in Section 6.3). The only requirements for this support, easily met by all embedded language structures we have encountered, are that they have well-defined linguistic boundaries and that their contents be tokenized as a single entity by the language lexer.[1]

This architecture permits utilization of any editors in

the JFC text framework, including the CodeProcessor recursively. The mapping from token types to editor types is performed by the Language Module; this module in the current prototype uses the standard JFC text editor for comments and a token-based CodeProcessor editor for strings and character literals.[2]

6. Functionality and user-model

This section presents an overview of the CodeProcessor's functional behavior as well as the user-model experienced by the programmer.

6.1. Advanced program typography

The CodeProcessor is visually distinguished by its advanced typographical "styles," implemented by the view architecture described in Section 5.3. These styles approximate designs by Baecker and Marcus [3] and are updated with each keystroke as the source code is being incrementally reanalyzed. Alternate styles for each language can be selected dynamically, either to suit individual preference or as required by particular tools driving the display. The style appearing in Figure 5 is configured
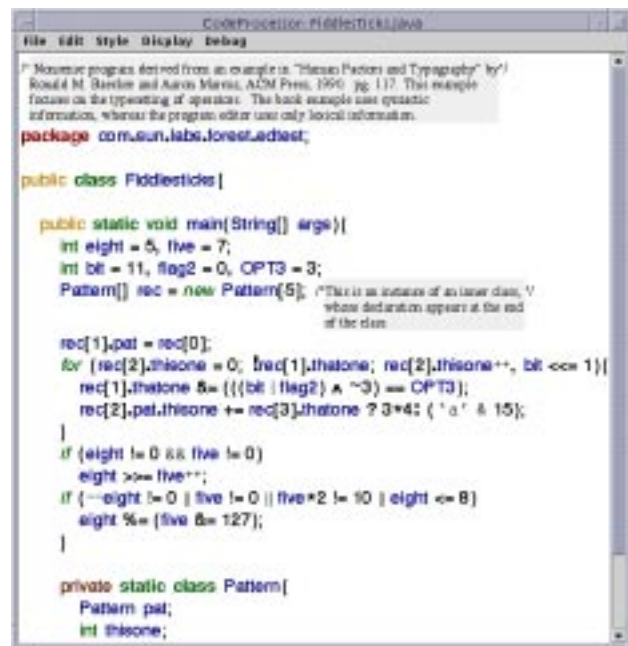

Figure 5: Example CodeProcessor display

by 123 token categories to which are assigned 61 separate token styles.[3] Each token style specifies type face, size relative to a base, style (plain, bold, italic), foreground and

---

1. If the nested editor is, in fact, another instantiation of the CodeProcessor, the contents of an embedded structure may be further tokenized by the nested lexer.

2. Both strings and character constants afford a simple lexical description that recognizes character escapes such as \n, \t, etc. This lets us, for example, highlight legal escapes so that they are distinguishable from the rest of the text, as well as indicate which ones are invalid.

3. Much of the stylistic detail is required as compensation for the absence of type faces suitable for programs [3].

background colors, baseline elevation, and both left and right boundary specifications used to compute display spacing between adjacent tokens. Token styles can also specify alternate display glyphs, for example to display ligatures.

In a departure from the Baecker and Marcus designs, which require well-formed programs, CodeProcessor styles reveal that certain tokens are lexically incomplete (for example "0x") or badly formed (for example "08"), based on lexical grammars extended to include such tokens. The CodeProcessor treats such tokens as legitimate in every other respect.

Although the Baecker and Marcus designs require full program analysis, a surprising amount of the visual detail can be achieved using only lexical information. Indentation requires fuzzy parsing in the style of many text editors. More visual features could be added through other kinds of fuzzy parsing, for example adjusting operator spacing based on expression depth.

Horizontal spacing between tokens is computed from the source code, not affected by presses on the space bar. This improves legibility and saves keystrokes, much in the same way that conventional auto-indentation works at the beginning of each line. We anticipate adding a tab-like mechanism to the current prototype that gives programmers some ability to impose vertical alignment.

6.2. Editing behavior

The CodeProcessor behaves like a code-oriented text editor in most respects. Where it differs, the behavior has been designed so that it appears to do the right thing when used as a text editor. Preliminary experience with the CodeProcessor's user-model suggests that programmers find descriptions of the behavior confusing, but the behavior itself unremarkable.

Some behaviors are completely conventional. Indentation is automatic. Line breaks are explicitly entered and deleted by the programmer.[1] Typing text within comments and language tokens (especially string literals) is likewise conventional, with the notable exception that programmers can easily type multi-line comments (and perhaps eventually strings), as shown in Figure 5.

Non-standard behavior appears in and around token boundaries. To first approximation, token boundaries are determined purely by the lexical analyzer. When the cursor rests between two tokens it is displayed midway between them; pressing the space bar silently does nothing.

However, not all boundaries can be unambiguously computed, for example between keywords. Here the CodeProcessor automatically inserts a "separator" token.

This behaves somewhat like a "smart space" in a word processor: no more than one can be present between adjacent lexical tokens. The cursor can rest on either side of a separator; deleting a separator is treated as a request to join surrounding lexical tokens (if they could not be joined, there would have been no separator present). Separators often come and go as the lexical categories of adjacent tokens are changed by editing, but since they are behavioral rather than visual, this is not distracting.

String literals and comments receive special treatment, as described in the following section. Additional subtleties in the user-model, beyond the scope of this paper, are required so that "the right thing" appears to happen at all times.

6.3. Nested editors

The user-model for editing programs described in the previous section is inappropriate in certain regions. The contents of string literals obey different grammars than surrounding code, and the contents of comments are not analyzed at all.

Such regions receive special support in the CodeProcessor, beginning with behavior that preserves their boundaries during all normal editing. This has the flavor of structure editing, but it solves a number of traditional problems with boundary confusion; potentially confusing behavior can smoothed over with careful design.

Having guaranteed boundary stability for these regions, the CodeProcessor can then provide specialized behavior in a straightforward way. Specialized editors are simply embedded to match the model: one kind for strings, another for character literals, yet another for plain text comments. More can be added, for example to support HTML or graphical comments. Although this has something of the flavor of a compound document system, it is specialized for source code and designed so that the boundaries are no more obtrusive than absolutely necessary. For example, the text cursor moves smoothly across boundaries between code and embedded structures.

6.4. The Programmer's Experience

The net result of these behaviors is by design an editing experience that is visually rich but otherwise unobtrusive. Nearly all familiar keystroke sequences have their intended effect, with the added bonus of fine-grained visual feedback. Time wasting efforts at whitespace management, for example deciding where to insert spaces and how to align multi-line comments, become as unnecessary as manual indentation. This frees the programmer to concentrate more completely on the task at hand: understanding and writing code. Furthermore, the rich display engine creates new opportunities for tools to present information by modulating the source code display to suit the task at hand.

---

1. The CodeProcessor does not break lines, but it would be helpful to add a linguistically driven mechanism for "wrapping" lines wider than the available window.

7. Implementation status

Initial design of the CodeProcessor was carried out at Sun Labs by the first author in the Spring of 1993. A prototype using C++, the *lex* analyzer, and the Interviews graphical toolkit [15], was demonstrated later that year as part of a larger programming environment project. An evolution of the first prototype, using the Fresco toolkit [7] (itself an evolution of Interviews) was completed and demonstrated in early 1995, at which time work ceased with the conclusion of the project. The design was then shelved, awaiting more suitable infrastructure than was available at that time.

The second author commenced a reimplementation of the CodeProcessor design during a summer internship at Sun Labs in 1998, adding recent improvements in incremental lexing technology and adapting the recently developed text framework from the JFC swing toolkit [30]. This prototype, which will be subject to further refinement and evaluation, is substantially complete, with the exception of automatic indentation and other services not part of the core design.

8. Related work

Emacs [23] is an augmented text editor of the kind described in Section 3.2. Its editing *modes* add specialized behavior and text coloring via pattern matching, but they fall short of the CodeProcessor's requirements. Weak encapsulation of its internal representation, as well as insufficient model-controller separation, makes reliable representation and manipulation of structural information difficult, if not impossible. Language analysis is limited to (unreliable) regular expression matching of fewer than ten lexical constructs. Rendering and layout, even in the more recent XEmacs [32], does not meet the CodeProcessor's demands. The editors embedded in many commercial integrated development environments have basic text editing and display functionality comparable to Emacs.

Numerous structure editors, mentioned in Section 3.1, were built in research environments, for example Centaur [6], Gandalf [18], Mentor [8], and PSG [4]. All had acknowledged usability problems [11].

The commercialized Synthesizer Generator [21] is a notable example of the modified structure editors described in Section 3.2, but was still plagued by confusing behavior [27] and by restrictions on editing.

The Pan system [5] is characteristic of the inclusive designs described in Section 3.3. It permitted unrestricted text editing, performed full incremental language analysis on demand, and provided semantic feedback. Although some attention was paid to usability [26], the implementation was enormously complex and offered no language-related advantages during textual editing. Important features such as comments received no special support at all.

Several elements of the CodeProcessor's design subsequently appeared in the Desert environment, including attention to usability, adoption of advanced typesetting, and the choice of a token-based representation [20]. FRED, the Desert editor, performs language analysis via integration with the FrameMaker document processing system [1]. This limits FRED's ability to support fine-grained language-based behavior due to the lack of appropriate abstractions in the Frame Developer's Kit API [2]. Moreover, reliance on a sizable document processing system reduces the likelihood of embedding FRED elsewhere.

9. Conclusions

We have designed and prototyped source code editing technology that addresses the full spectrum of requirements faced by designers of software engineering tools. This technology matches programmers' skills and expectations, and brings to bear the power of language-based technology in support of both the people and other tools in the environment. Meeting these often conflicting requirements required both a new user-model for its behavior as well as a new architecture. Its construction stretches the limits of the existing infrastructure.

History tells us that less ambitious designs will fail. Some language-oriented technology can be grafted onto simple text editors, but insufficiently rich representations limits their power and accuracy. Some usability compromises can be made to language-oriented structure editors, but the fundamental architecture dooms their usability.

A lexical-based architecture by itself would also fail, since a naive user-model would suffer many of the ills of tree-oriented editors. Likewise, the new user-model by itself would fail, since the mismatch between it and existing representations would preclude adequate implementations.

The CodeProcessor performs enough linguistic analysis to permit useful tool integration, as well as useful language-based services such as high-quality on-the-fly typography. At the same time its fundamental behavior is textual, permitting easy adoption by programmers, and it includes specialized support that simplify and extend comment management significantly.

Designing tools that are both powerful and effective is difficult, and the more "low level" the tool, the more demanding are the user requirements. Starting with these requirements, however, and embracing the notion that powerful tools must above all fit with programmers skills, expectations, and tasks, gives hope that benefits of software development technology can actually make a difference in the way people work.

10. Acknowledgments

this paper, and we thank the anonymous reviewers for their constructive suggestions as well.

11. Trademarks

Sun, Sun Microsystems, and Java, are trademarks or registered trademarks of Sun Microsystems Inc. in the United States and other countries.

References

[1] Adobe Systems Incorporated, Adobe FrameMaker, http://www.adobe.com/products/framemaker/

[2] Adobe Systems Incorporated, Frame Developer's Kit, http://partners.adobe.com/asn/developer/framefdk/fdkguide.html

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

[4] 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.

[5] 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.f

[6] P. Borras, D. Clément, 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.

[7] Steve Churchill, "C++ Fresco: Fresco tutorial," *C++ Report*, (October 1994).

[8] 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.

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

[10] 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.

[11] 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

[12] 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.

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

[14] 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.

[15] Mark A. Linton, John M. Vlissides, and Paul R. Calder, "Composing user interfaces with InterViews," *Computer*, **22**,2 (February 1989), 8-22.

[16] Metamata, Inc. "JavaCC - The Java Parser Generator: A Product of Sun Microsystems," http://www.metamata.com/JavaCC/

[17] 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.

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

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

[20] Steven P. Reiss, "The Desert Environment," *ACM Transactions on Software Engineering and Methodology* **8**, 1 (October 1999), 297-342.

[21] Thomas Reps and Tim Teitelbaum, *The Synthesizer Generator Reference Manual*, Springer Verlag, Berlin, 1989. Third edition.

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

[23] 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.

[24] Gerd Szwillus and Lisa Neal (editors), *Structure-Based Editors and Environments*, Academic Press, 1996.

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

[26] 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, reprinted in [24].

[27] Michael L. Van De Vanter, "Practical Language-Based Editing for Software Engineers," 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.

[28] Tim A. Wagner, *Practical Algorithms for Incremental Software Development Environments*, UCB/CSD-97-946, Ph.D. Dissertation, Computer Science Division, EECS, University of California, Berkeley, December 1997.

[29] William M. Waite and Gerhard Goos, *Compiler Construction*, Springer-Verlag, 1984.

[30] Kathy Walrath and Mary Campione, *The JFC Swing Tutorial: A Guide to Constructing GUIs*, Addison-Wesley, 1999.

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

[32] XEmacs, http://www.xemacs.org