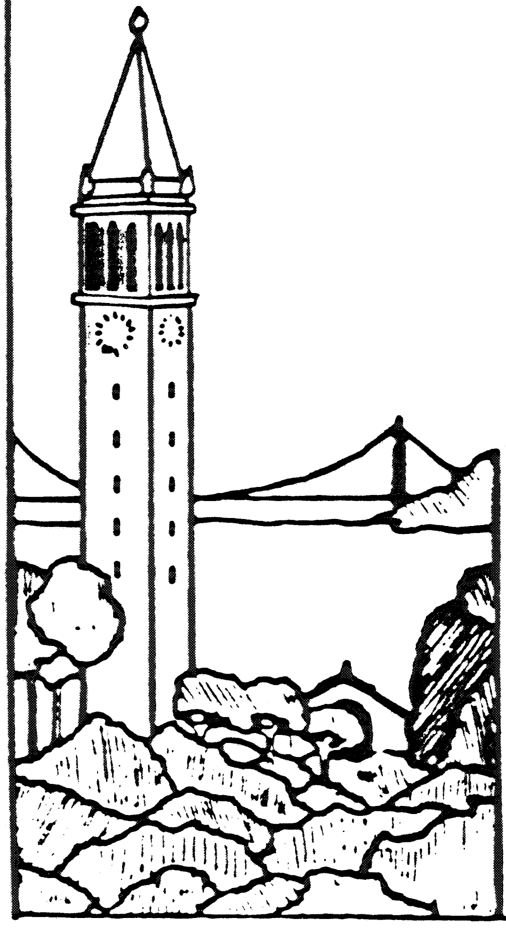
User Interaction in Language-Based Editing Systems

Michael Lee Van De Vanter



Report No. UCB/CSD 93/726

December 1992

Computer Science Division (EECS)
University of California
Berkeley, California 94720

User Interaction in Language-Based Editing Systems

Michael Lee Van De Vanter

Computer Science Division—EECS
University of California, Berkeley
Berkeley, California 94720

December 1992

Dissertation submitted in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy in Computer Science
in the Graduate Division of
the University of California at Berkeley

Report No. UCB/CSD-93-726

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. The content of the information does not necessarily reflect the position or the policy of the U.S. Government.

User Interaction in Language-Based Editing Systems

Michael Lee Van De Vanter

Abstract

Language-based editing systems allow users to create, browse, and modify structured documents (programs in particular) in terms of the formal languages in which they are written. Many such systems have been built, but despite steady refinement of the supporting technology few programmers use them today. In this dissertation it is argued that realizing the potential of these systems demands a *user-centered* approach to their design and construction. *Pan*, a fully-implemented experimental language-based editing and browsing system, demonstrates the viability of the approach.

Careful consideration of the intended user population, drawing on evidence from psychological studies of programmers, from current software engineering practice, and from experience with earlier systems, motivates *Pan*'s design. Important aspects of that design include functional requirements, metaphors that capture the feel of the system from the perspective of users, and an architectural framework for implementation.

Unlike many earlier systems, *Pan*'s design hides the complexity of language-based technology behind a set of simple and appropriate conceptual models — models of the system and of the documents being viewed. Responding to the true bottleneck in software production, *Pan*'s services are designed to help users understand software rather than save keystrokes writing it. Furthermore, *Pan*'s design framework provides services that degrade gracefully in the presence of malformed documents, incomplete documents, and inconsistent information.

This research has yielded new insight into the design problem at all levels: the suitability of current language-based technology for interactive, user-centered applications; appropriate kernel mechanisms for building coherent user services; new conceptual models of editing that blend textual and structural operations without undue complexity; and the crucial role of local, site-specific design in the delivery of language-based editing services.

To the memory of my mother

Helen Carol Van De Vanter

Acknowledgements

Pan is the work of many people. Special thanks go to Robert Ballance and Jacob Butcher, from whose collaboration many of *Pan*'s good ideas emerged and were realized. They also built *Colander* and *Ladle* respectively, the two crucial parts of *Pan*'s language-based infrastructure. Important contributions also came from Christina Black, Laura Downs, Bruce Forstall, Mark Hastings, Darrin Lane, William Maddox, John Pasalis, Tim Wagner, and Robert Wahbe.

Yuval Peduel's efforts at keeping my thinking and writing clear have been a great help, as have the support and encouragement of colleagues Gordon Hamachi, Peter Kessler, Paul Kube, Jock MacKinlay, Vicki O'Day, Eduardo Pelegrí-Llopart, Richard Probst, Dain Samples, and Polle Zellweger.

I would like to thank my committee members, Susan Graham, Michael Harrison, and Andy diSessa, for their guidance. My research advisor, Susan Graham, deserves special thanks for her wisdom and foresight throughout the project.

Finally, I would like to thank my wife Karen for her patient support, and my children Rebecca, Garth, and Justin for being themselves.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software systems are unique among engineered artifacts: to commit a detailed design to a suitable formal notation is to construct the system being designed. The natural preoccupation with notation in software engineering motivates a corresponding interest in tools that help people create and manage increasingly complex software documents.

Language-based editing systems represent an important evolutionary thread in the development of tools for the construction of software. These systems allow users 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.

One 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 [7,17,29,94,104], and practical systems of this kind are now within reach, for example the single-language SMARTSystem[1] for C [97].

Growing experience with these almost-practical systems suggests that more attention now be given to another part of the problem: user interaction. This need goes 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 professionals 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.

Asking user-centered rather than technology-centered questions casts new light on the design of these systems, their internal software architecture, services offered to users, configuration mechanisms, styles of interaction, and integration with other tools in the working environment. It also raises new questions about the underlying language-based technology, its usage, and what problems remain to be solved.

[1]SMARTSystem is a registered trademark of the ProCase Corp.

1.1 Thesis and Scope of This Research

The thesis of this investigation is 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. When reconsidered from this perspective (drawing on evidence from software engineering practice, human factors, and empirical experience), many usability problems can be seen in past systems:

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.

Each of these problems has been partially addressed by earlier systems, but usually at the expense of other problems on the list. The tradeoffs are complex, however, and successful solutions must take the entire list into account.

This investigation begins with the following observation about these usability problems:

> 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 user and software documents (the more important meaning in this context).

With the design challenge posed in this way, new questions must 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 [92] 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.

This dissertation presents a new design framework for language-based editing systems, one that begins with a basic separation of design concerns: internal document representation and analyzer implementation; configurable, language-independent mechanisms to support user interaction with documents; a coherent and flexible set of user-visible system features and policies; and adaptation of the system to particular working contexts. *Pan* is a fully implemented, multi-lingual, language-based editing and browsing system developed at the University of California, Berkeley [12,31,132] that embodies this framework and demonstrates the viability of the approach.

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. Users do not think of programs in the same terms that are useful for designers of compilers. For example, a person confronted with a new program first reads comments and then examines the names of program entities [20], for example procedures and variables. A compiler confronted with 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: (a) includes traditional syntax and static-semantic language descriptions, but may extend to extra-lingual analysis such as stylistic and usage guidelines; (b) specifies services to be provided and specializes generic services for the particular language; (c) defines a visual context, including typography and use of color; and (d) 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 users. 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, and debuggers.

1.2 Overview of Dissertation

Chapter 2 begins with an introduction to the context of this research: the intended users, the class of tools involved, and the ultimate role to be played by such tools. The chapter then reviews the list of usability problems identified above, bringing outside perspectives to bear on the history of language based editing systems. The software engineering perspective casts light on the working environment and the nature of software systems. The individual perspective examines what is known about the mental processes of the people who carry out those tasks. From the list of shortcomings are derived a set of proposed requirements for language-based editing systems that will correct those problems, requirements addressed by the prototype system designed and built for this research.

Chapter 3 introduces *Pan*, beginning with a few short scenarios that show it in action. An introduction to *Pan*'s design framework follows: guiding principles, a four-layered architectural model, and five design metaphors that capture the spirit of the system from the perspective of its users.

In Chapter 4 the design and construction of *Pan* are described in more detail, addressing in turn each layer in the architectural model, beginning with the lowest: (1) language-based infrastructure; (2) kernel mechanisms; (3) elements of user interaction; and (4) view style design. This model enables an implementation that delivers language-based functionality without allowing the design of the analyzers to dictate models for user interaction. This chapter presents a "structural" description of the system, reflecting a point of view concerned with implementation: how the system is constructed and how it works. Sections 4.1 and 4.2 focus on internal issues, Sections 4.3 and 4.4 on those aspects presented to users.

Chapter 5, in contrast, represents a "functional" description of *Pan* that focuses on users and their tasks: what the system can do and how it can be used. Each section examines a particular aspect of user interaction: potential benefits with reference to the goals discussed in Chapter 3 and challenges to effective implementation. Some of the challenges result from conflicting goals, others from particular language-based technology. In each case *Pan*'s framework provides mechanisms, services, and design guidelines to solve the problems in accordance with general principles of user-centered design.

- **Text Presentation** An important step toward enhancing program comprehension by users is to exploit the full potential of human visual bandwidth; Section 5.1 describes how this can be done using a high-quality, configurable, text-based display. A unifying theme for *Pan*'s text-based services is to provide as many visual channels as possible for the conveyance of *meta-information*, that is, information *about* programs being viewed.

- **User Models** Section 5.2 turns to problems that arise when browsing and editing services operate over a domain whose structure is not isomorphic to the visual field presented to users. This is a particular problem for language-based editors, where the customary approach is to display programs in the conventional two-dimensional textual field, but to provide operations over the domain of a *tree* that represents an abstraction of a program's formally-specified context-free syntax. *Pan*'s design supports a model that more closely approximates how programmers actually think about program structure, thereby avoiding many usability problems associated with earlier systems.

- **Mixing Modes** A commitment to unrestricted text editing combined with language-based operations runs the risk of exacerbating the problem of hidden state: information in the dynamic editing context that is important to the user, but which has no direct visual analogue. These problems are minimized in *Pan*'s design by presenting to the user a model of language-based services that appears to be a simple extension of the familiar text-based model, accompanied by a metaphor to explain the relationship between the two. A *Pan* user may mix text-based and structure-based operations freely. Experience with this model, combined with further task-based analysis, has led to a novel design for *structural selection*; the new model promises to simplify user interaction and to permit convenient delivery of many useful services not otherwise possible.

- **Ill-Formedness** Unrestricted text editing, an important commitment in *Pan*'s design, implies that a program being modified is often ill-formed with respect to the definition of the formal language in which it is written. Many language-based editing systems treat these as aberrant situations, and they restrict or otherwise penalize the user until the problems are corrected. In *Pan*'s design ill-formed programs are presumed to represent the normal case, and services are designed to keep operating as usual. In fact, it is crucial that one distinguished set of services, those concerned with diagnosing and communicating about language errors, continue to be available whenever the user requests them, as described in Section 5.4.

- **Incompleteness** Section 5.5 shows how incompleteness, a special case of ill-formedness, can be addressed by *Pan*'s variant on strict structure-based or top-down editing: *placeholders* and *templates*. Like all *Pan*'s services, these add to the user's options rather than restricting them.

- **Inconsistency** A more subtle problem with the commitment to unrestricted text editing arises in the form of difficult choices about when and how to attempt incremental reanalysis following textual modifications. These choices involve issues as diverse as the cognitive structure of the user's tasks, the ability of analysis algorithms to recover from program errors, and the speed of computational platforms. The approach taken in *Pan*'s design is to adopt a lazy policy, deferring analysis until the user requests it, either explicitly or implicitly, by invoking an operation that requires current information. In the interim, between a textual modification and a subsequent reanalysis, the program state in *Pan* is said to be *inconsistent*. Section 5.6 describes several approaches that help make this state tractable for users.

- **Views** Section 5.7 shows how more than one *view* of documents can be useful to users. An alternate view in *Pan* typically presents some subset of the information available about a document, organized and presented in a form helpful for particular tasks. Connections among all views of a single document, however, permit the user to associate the different kinds of information when needed.

- **Applications** In Section 5.8 a number of more advanced services are described. These services have been or could be implemented using the basic elements of *Pan*'s design. The breadth of these applications demonstrates the flexibility of *Pan*'s model for language-based interaction.

A reflection on the success of the *Pan* prototype in meeting its research goals appears in Chapter 6. For example, some aspects of user interaction have shown themselves to be unsatisfactory in the current prototype, based on both experience with the system and ongoing analysis. An important kind of insight gained through work on *Pan*'s infrastructure concerns the degree to which language-based mechanisms must be recast in pursuit of user-centered goals.

Finally, Chapter 7 summarizes the lessons learned during this research, and includes suggestions concerning promising areas for further work.

Chapter 2

Background

This chapter introduces language-based editing, with particular attention to the context in which these systems are intended to be used. Evidence about that context, drawn from software engineering practice and from the study of computer human interaction, confirms empirical observations that usability problems abound. These problems are discussed here in general terms, and requirements are articulated for a system that corrects the shortcomings mentioned in the introduction.

2.1 Context

This section describes the context for this research: the intended user population and the class of tools being studied.

2.1.1 The User Audience

Software professionals manage large collections of interrelated documents. Many traditional assumptions about traditional language-based editors do not hold in this domain, for example that the main advantage of such systems is to help beginners avoid syntax errors, and that restricting users is a reasonable cost to pay for that advantage.

In terms of Neal's model of expertise among users of program editors [90], these professionals have high levels of expertise with computers, and usually have high levels of expertise with both programming and with languages. There are situations in which these users share some characteristics with beginners, for example when learning a new language, but it is important that design compromises not be made to accommodate low levels of general and programming expertise.

The working context is further enriched by familiarity with many existing tools, their functionality and their interface characteristics. New tools will be rejected if they merely duplicate existing functionality, if they operate needlessly unlike others, or if they cannot share data with other tools.

2.1.2 Language-Based Editing Systems

The term **language-based** indicates that one or more system facilities exploit language-specific information derived from documents. In the context of this research, the term

system (or **editing/browsing system**) encompasses all services used to browse, manipulate, and modify one or more documents interactively.

In the broad view there are many such systems, beginning with text editors that support word- and paragraph-based operations, including spelling checkers, and continuing through document preparation systems and outline processors. The success of spelling checkers (measured by their widespread use) offers a suggestive analogy for the design of language-based tools: they usually operate only when asked, and they carefully defer to the judgment of the user. The implicit assumptions are that a spelling checker is clerical but fast, must be given a dictionary, and cannot be assumed to understand anything about the *intent* of the author.

For software documents, linguistic knowledge available to the system can be shallow (for example language-modes in EMACS exploit shallow lexical structure) or deep (for example *cliché*-based editing in KBEmacs, which exploits deep plan-based structure [106]). At the limit of currently feasible technology are systems that have knowledge of the syntactic and static-semantic structure of software, and perform some browsing and manipulation using that information.

Proponents of language-based editing appeal to the compelling intuition that a system knowledgeable about underlying languages could help users with many of the demanding aspects of programming. Bernard Lang, coauthor of the influential Mentor system bases his testimonial on considerable experience with Mentor: "For advanced users, a syntax directed environment is irreplaceable for program maintenance, especially when used to the full capacity of this technology, with the ability to use all tools and program new ones in terms of the syntax directed paradigm, i.e. on the basis of the abstract syntax of the manipulated languages" [71].

At the time this research began, a number of language-based editing systems had already been implemented or designed. Following Ballance's taxonomy [9], those can can be roughly classified into three categories based on underlying models of editing: display-oriented text editors such as EMACS [122], syntax-directed structure editors such as Mentor [29], and syntax-recognizing editors such as Babel [57]. The rest of this section describes the three categories in more detail, pointing out the design tradeoffs they represent. Each category in the list reflects progressively more ambition with respect to services delivered, along with correspondingly greater design challenges.

How a system captures and maintains structural information involves an important distinction. It can be done *explicitly*, by interpreting user actions during document construction, as in an outline processor where the user creates new sections explicitly. Alternately it can be *derived* by examination of the document after construction by the user.[1] The choice affects both underlying implementation and the design of the user interface. For example, *syntax-directed editors* rely on the user to construct structures explicitly. *Syntax-recognizing editors* attempt to derive a structural representation for the document. A purely *text-oriented* editor makes no real use of linguistic structures at all; its "language-based" operations examine only the surface representation of a document, perhaps inferring a very simple syntactic decomposition.

[1]The two approaches can coexist; for example Mentor can derive structure initially from a textually represented document, but subsequent editing can only be structure-based [29].

Text-Based Editors

A **text-oriented editor** operates on a document modeled both as a stream of characters and as a two-dimensional plane of characters. (The coordinates in this plane are commonly called "lines" and "characters within a line".) Users are free to operate on any character at any time.

While groups of characters, for example words and lines, might be accorded special status and operations, no special structural constraints are imposed on the document and few well-formedness constraints are enforced. Bravo [70], EMACS [122], MacWrite [2], and Z [140], are examples of text-oriented editors. Both EMACS and Z provide some language-specific editing operations.

The technology for text-oriented editors is relatively mature. Language-based editing operations in these systems present no great technical obstacles, but the level of service is quite limited.

Structure and Syntax-Directed Editors

Structure editors present a document as having a definite internal structure, with editing operations modeled as operations upon that structure. Most often, the document is tree-structured, with operations defined on subtrees. Outline processors are structure editors for tree structures; spreadsheet editors are structure editors for tables. Structure editors may or may not interpret the structures that they edit. When they do attribute language-specific meanings to the structures, they are often called syntax-directed editors.

A **syntax-directed editor**[2] is a structure editor that requires that the document be syntactically correct at all times: editing operations must "follow" the syntax of the language. As in a pure structure editor, the user can add new material to a document only at those points where it can be successfully grafted onto the existing structure. ALOE [84] and the Gandalf editors [94], Centaur [17], the Cornell Program Synthesizer [125] and the Synthesizer Generator [104], Emily [52], Mentor [29], PSG [7], and the SbyS editor of the Mjølner project [87] are all syntax-directed editors.

Structure and syntax-directed editors are considerably more complex to build than text-oriented systems: language-based machinery must be added, and a complex relationship must be maintained between text-based visual presentations and internal representations more amenable to language-based analysis. On the other hand, design of these editors is simplified by insisting that users express changes in terms of internal representations rather than in terms of the textual presentations.[3]

Syntax-Recognizing Editors

A **syntax-recognizing editor** [21] derives structural information from text, in order to check for correctness without necessarily demanding the consistency constraints of a syntax-directed editor. Syntax-recognizing editors can provide structural operations as well as text-oriented editing. Babel [57], the Saga editor [67], SRE [21], Syned [40], and the UQ editors [138] are all syntax-recognizing editors, as is *Pan*, the platform for this research.

[2]The term "structure-oriented" [93] is sometimes used in place of "syntax-directed."

[3]Some *hybrid* syntax-directed editors, including the Gandalf editors, the Cornell Program Synthesizer, and PSG, permit text-editing in limited contexts.

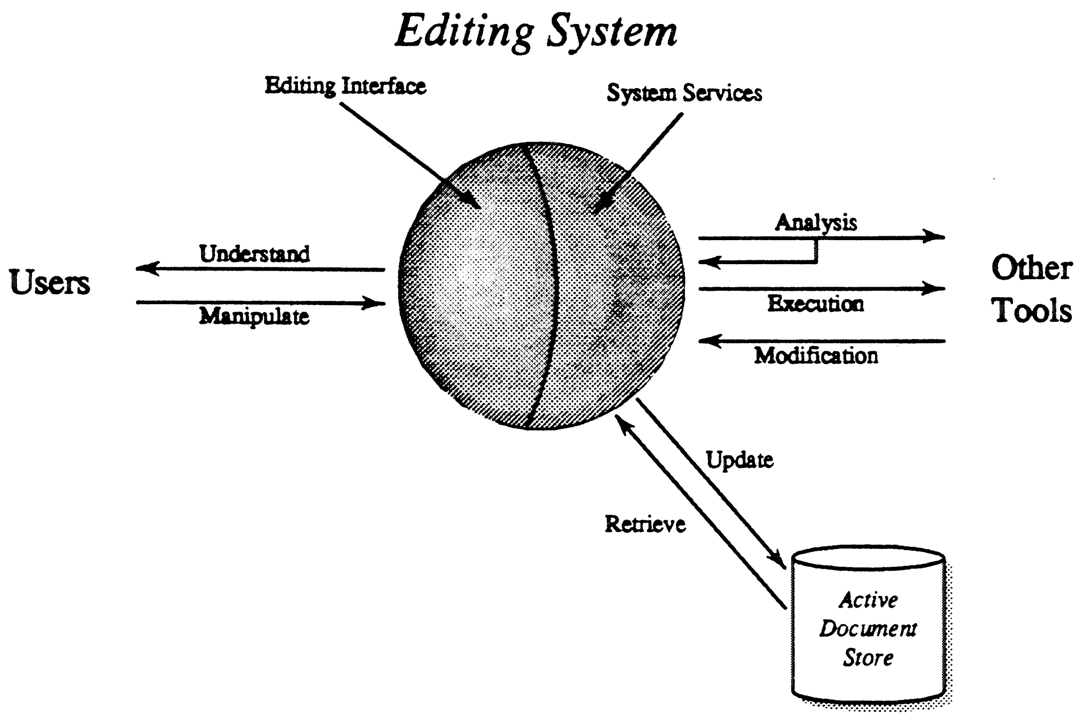


Figure 2.1: Editing interface and system services in relation to the environment

The syntax-recognizing approach[4] permits text-oriented editing by users at any time in any context, but at the cost of considerable complication in document representation, incremental analysis algorithms, and user-interface design. Common problems in early syntax-recognizing editors were a failure to hide this complexity from the user and limitations on language-based services when compared with syntax-directed editors.

2.1.3 The Role of The System

Early language-based editing systems were characteristically designed as standalone tools, although a few were integrated with a debugger and run-time system. In practical software engineering contexts a shift of perspective is necessary. The true utility of such tools is as the primary *interface* between people and integrated environments containing the documents they manage. Positioned this way, between user and documents, the system is uniquely situated to share information *about* documents that may be provided by both user and tools, as shown in Figure 2.1.

In this model, users interact with documents through the editing interface; tools interact with documents through the system services; they communicate with one another via an active data repository. Useful tools in this model are characterized by information, possibly rapidly changing, *about* the code. Examples include debuggers, slicers, profilers, text coverage analyzers, and higher-level assistants like the cliché-based reasoning supported by KBEmacs [106].

[4]The syntax-recognizing approach does not preclude a user-interface that simulates syntax-directed editing. In fact, syntax-directed editing can be provided easily in a syntax-recognizing editor.

Complex, expensive analyses to support an editing interface make sense only in an environment in which many tools share the information maintained by the system. The same type checking that is used to tell the user that a document is type-correct can provide type information to a compiler, an interface consistency service, or an auditing tool. In some cases information produced by other tools should be made available to the editing interface. For example, the results of performance analysis or information derived from version control can be used to produce helpful views of programs or prototypes.

2.2 Usability Problems

This section returns to the list of shortcomings identified in the introduction, explaining the origin of each and associated difficulties for users. From the discussion emerge the design requirements pursued during this research.

2.2.1 Text-Oriented Interaction

Usability Problem 1 Annoying restrictions on text-based editing, the style of interaction most people prefer.

The history of language-based editors is marked by disagreement over text-oriented editing and the role it should play in language-based editing systems. Text remains the display medium of choice for documents in most languages, since most languages are by definition textual. Graphical presentations of such documents have some potential value for overviews and summaries, but it is very unusual for a graphical display to carry *all* the information present in a textual rendering, and it is seldom a natural visual field in which users can specify editing operations.[5]

The most fundamental requirement for language-based editing systems, discussed in this section, is thus the most controversial:

Design Requirement 1 Familiar, unrestricted text editing.

The Structural Hypothesis

Following the lead of the purely structural Emily system [52], one of the seminal language-based systems was designed around what will be called here "the structural hypothesis:"

> 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 [125].

Despite a few early arguments to the contrary [134,140] designers of language-based editors accepted the hypothesis and continued to impose varying degrees of restrictions on text-based interaction. Most designers argued that prohibiting "incorrect" operations would help users write programs faster or that it would enable the system to provide better services.

[5]This remark does not apply, of course, to those languages which are by definition graphical. For those languages, graphical display is the medium of choice.

One problem with the structural hypothesis is that language-based structure is not uniquely defined for a given language, but reflects instead a set of implementation choices based loosely on a formal language definition. For example, many early editors that provided unrestricted text editing maintained a complete parse tree [57,67], and there are many degrees of freedom available for the design of parsing grammars. Syntax-directed editors, on the other hand, usually maintain a reduced or **abstract** syntax tree (an example of each appears in Figure 2.2 on page 15). Arguments for the latter representation include the observation that it is a more "natural" model for user interaction, since the extra information in the parse tree is only an artifact of parsing technology. In practice, however, abstract tree representations are designed to meet the needs of tool implementation, not people.

A second problem with the structural hypothesis is that every operation burdens the user with the cognitive overhead of a complex relationship between the tree and its two-dimensional textual presentation. Lang mentioned evidence of this problem with Mentor:

> We must however note that, even with a good user interface, the syntax directed paradigm is too complex and bothersome for inputting programs or for performing simple editing tasks. As might be expected, it is particularly inconvenient for editing text/program fragments that are non-structured (strings, comments) or poorly structured (expressions) [71].

This kind of experience with fine-grained structural interaction led to the adoption of the hybrid approach by a number of systems, including the Cornell Program Synthesizer, in which small structural fragments may be edited textually.

Behind this second problem is serious clash between the editing model (what the user can do) and the presentation model (what the user sees). One can imagine a similar clash arising from a text editor that required sentences to be constructed only in terms of an underlying natural language grammar. Minör's attempt to correct this problem without abandoning the structural hypothesis led away from a text-oriented interface entirely for the SbyS program editor [87,88].

A third weakness is the implicit assumption that the only interesting structural categories are those corresponding to nonterminals in an underlying abstract grammar. But programmers' tasks involve many kinds of information beyond the purely syntactic, as will be discussed in more detail below.

Experience with Structural Editing

Experience confirms these arguments against the structural hypothesis. For example implementors of Rita, an editor for structured documents, were forced to augment their purely structure-oriented user interface with text-oriented operations because:

> Many editing operations which appeared simple and straightforward from the user's perspective, could involve fairly complex keystroke sequences because of the nature of tree-processing operations. Although most users understand that there are rules governing the structure of a tagged document, they tend to view a document as a linear progression of text rather than as a hierarchy of document elements [27].

Authors of two important language-based system for programs, PSG and Gandalf, have likewise concluded that text-oriented editing should not be limited [8,24].

Lisa Neal examined how people react to this kind of system, concluding that "very experienced users will reject any tool that does too much for them. For instance, they will only use a tool which allows free text input but provides error checking capabilities" [90].

What Users Want

In practice, people will not sacfifice text-oriented interaction. It fits naturally with the visible nature of the notation, and people are accustomed to it. Furthermore, most kinds of documents contain textual chunks that have no structural properties beyond the textual: examples include sentences or paragraphs in most natural language documents, labels in spreadsheets, and comments in programs. Finally, in a world of multiple languages text remains the least common denominator, a fallback position for users that is necessary when language-based systems produce different modes of interactions for each different language.

2.2.2 The Comprehension Bottleneck

Usability Problem 2 Failure to address the real productivity bottleneck, the difficulty people have understanding programs.

Early arguments for language-based editing systems focused on the need to expedite entry of syntactically correct programs. For example, one advantage cited for grammar-based template expansion is that it saves keystrokes. The COPE system uses *predictive data entry* during textual entry; it performs very elaborate error repair that allows the user to enter somewhat abbreviated code [4]. Unfortunately systems tuned for expeditious entry fail to address real obstacles to productivity.

Mary Shaw, reflecting on software engineering, observed that "Each order of magnitude increase in the scale of the problems being solved leads to a new set of critical problems that require essentially new solutions" [114]. An important example is that software systems have become so large and complex that developers spend far more time trying to read, understand, modify, and adapt documents than they do creating them in the first place [45, 139]. This observation leads to the next requirement for language-based editing systems:

Design Requirement 2 Rich information display.

Visual Presentation

One approach to enhancing comprehension draws on established traditions of graphical design in order to exploit the full power of the textual medium. The value of high-quality typography for natural language documents is well established, and recent studies suggest the same potential benefits for programs [6,95].

These studies, however, involved printed versions of programs typeset onto paper from source code using language-specific formatters, a context that differs in two ways from that of language-based editing systems. First, color produced by CRT displays evokes different perceptual effects than it does on paper [33], and other characteristics of the CRT medium have observably detrimental effects on reading speed [47].

Dynamic Display

A second difference between paper and a language-based system is the dynamic nature of the context: additional kinds of information (cursor location and the like) must be displayed, and users can potentially control what information gets displayed. The user's needs for dynamic display can be seen by considering in more detail what it means to read software.

Programmers read because they spend most of their time engaged in software maintenance, where the essential task is understanding existing software; this is called "design recovery" when some of that understanding is to be recorded [15].

Evidence suggests that reading software is a cognitively active process and has a fine-grained task structure [68]. The reader repeatedly forms hypotheses, which are then confirmed or denied by further reading. This happens opportunistically, using a variety of information when it is available. Reasoning tends to alternate between forward-chaining and backward-chaining, depending on the information available [74]. This suggests that a language-based editing system's display be visually rich and dynamic at the same time.

Even when writing new documents, however, programmers spend most of their time reading what they've just written. This is an essential characteristic of the notation and the process of writing in it. 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. One finds similar recursive interaction between author and expressive medium in many creative endeavors [60], as well as in the work of Schön's "reflective" practitioners [110]. So one aspect of the author's conversation with the notation is a stream of questions like "Where am I now?" "What are the implications of what I've done so far?" and "What's left to do?" This argues all the more strongly for a rich, flexible information display (more about this below).

2.2.3 Technology Exposure

Usability Problem 3 Exposure of underlying language-based technology to people, leading to inappropriate user models of system state and document structure.

Some of the difficulty users encounter with language-based editing systems can be traced to inattention to principles of user-centered system design [92]. In particular such systems often fail to present a coherent conceptual model of system behavior that helps users apply the system to their tasks.

Given the role proposed earlier for language-based editing systems, as interface between user and software documents, it is equally valid and ultimately more important to apply the same principles in a second way. Coherent interaction between users and documents can be achieved by presenting conceptual models of document structure that help users understand them. This section looks at both kinds of interaction, with document structure and with system state, and sets forth the next requirement for an effective language-based editing system.

Design Requirement 3 Coherent user interaction with system and programs.

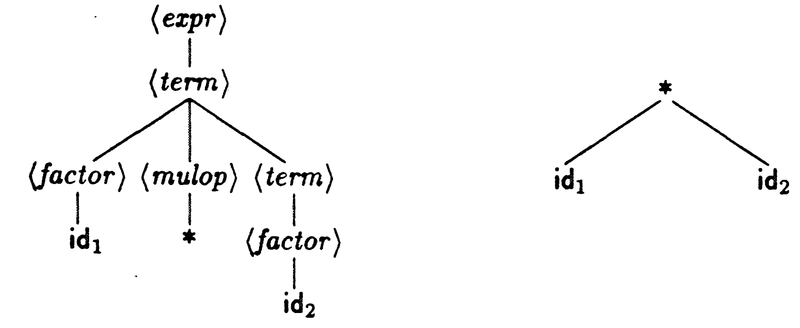


Figure 2.2: Parse versus Abstract Tree Representations

The Structure of Software Documents

Making language-based structure intelligible requires that the system present document structure in terms of coherent conceptual models. Most language-based editors present a document model based implicitly on the way they represent documents internally, typically as a tree. For example, many early editors that provided unrestricted text editing maintained a complete parse tree [57,67], whereas syntax-directed editors usually maintain a reduced or **abstract** syntax tree (an example of each appears in Figure 2.2). Arguments for the latter representation include the observation that it is a more "natural" model for user interaction, since the extra information in the parse tree is only an artifact of parsing technology.[6] Some of the problems with this approach were mentioned earlier in the critique of the structural hypothesis, namely that typical internal tree representations are not well-defined by languages, that they can cause confusion when used as the primary editing model, and that they don't correspond to the user's ideas of relevant structure. Consider for example the parent node ⟨*expr*⟩ of the parse tree in Figure 2.2 and its child ⟨*term*⟩. Both correspond to the expression that the user sees as "id_1 * id_2", the distinction is only an artifact of parser and perhaps analyzer technology, and simple structural navigation from one to the other would produce no apparent change in location.

From a user's perspective software documents are richly connected, overlapping webs of information having many structural aspects. Each aspect is more relevant for some kinds of users than for others and for some tasks more than for others. A system that supported such a multiplicity of structural aspects could confuse users, but it need not if the system supports those structures already understood by users. People routinely think about complex objects from different perspectives and are remarkably adept at shifting perspective. An effective editing interface need only support these shifts without imposing any extra overhead on the user.

The formally defined syntax of a language, and closely related internal tree representations, represents a good starting point for exploring aspects of software structure. It provides a "backbone" decomposition of documents into structural components that help the system associate useful properties with regions of text on the screen. Even here, however, many aspects of an internal representation (many kinds of tree nodes) may *not* correspond to any useful conceptual structure.

[6]An abstract tree also requires much less storage than an equivalent parse tree, a difference that can translate into improved system performance.

More interesting structures tend to involve non-local and non-hierarchical relationships among document components. For example, consider the relationships in a natural language document defined by the connection between a figure and references to it, the relationships between declarations (definitions) and uses of variables in a computer program, the structures represented by a call graph in a program, or all uses of a particular code library. These are examples of relationships that can be derived by language-based analysis, and which can then be used for browsing, for querying, and by treating them as links for the purpose of non-local navigation in the style of hypertext.

Other decompositions can arise from non-derivable design information that the user adds. For instance, editors like ED3 [123] and the Cedar editor Tioga [127] allow users to specify explicitly a hierarchical decomposition for each document, where the structure may or may not be correlated with structures in the underlying formal language. Outline processors are editors for precisely this kind of structure where the underlying language is simple text.

Some kinds of decomposition fall into the area between linguistic and non-linguistic. Such structure typically concerns how a language is used, and derives from the design of particular systems and from coding conventions they use. These carry a great deal of the information that a programmer needs, and may be partially derivable, for example, using analyzers that can identify uses of locally proscribed language features, and can reason about stylized patterns of use. These kinds of analysis move far beyond simple error checking supported by earlier language-based systems: they involve knowledge of particular organizations, techniques, and systems.

User Models of System State

Just as coupling the user's model of document structure too closely to underlying implementations can make documents hard to understand, confusing behavior can result from excessive coupling between important aspects of a system's interactive state and its internal representations. Here user-centered design of a more conventional sort can help. For example, it is important to make visible those aspects of the system's state that have direct consequences for user interaction. An example from the Synthesizer Generator Reference Manual [102, page 91] shows what can happen otherwise:

1. The user sees "`((2 + 3) * 4)`" and highlights (places the structure cursor at) the inner expression "`(2 + 3)`";

2. Editing textually within the focus (structure cursor), the user deletes the parentheses, leaving "`2 + 3`" in the focus.

3. The user invokes the command **`forward-with-optionals`** to move the structure cursor away from the change site; and

4. The system re-inserts the parentheses before moving the cursor.

This behavior is confusing and probably not what the user intended, hence a warning and this example in the manual. Just before the user invoked the navigation command in step 3, only a subtle line of reasoning (which involves taking careful note of the structure cursor placement) would help the user predict from the display "`(2 + 3 * 4)`" what would happen next. Had the structure cursor instead been placed during step 1 over the outer expression "`((2 + 3) * 4)`", the system would have done what the user probably intended; even then, however, new parentheses would mysteriously appear around "`3 * 4`". The problem in this case is that fine-grained details of the (invisible) internal representation have been allowed to exert influence over the system's behavior.

A related problem arises in systems that support structural cut and paste operations. For example it might seem reasonable to copy the list of identifiers appearing in the formal parameter list of a procedure definition and then to paste it into a call to that procedure. Although the two lists of identifiers might be textually identical and closely related conceptually, there may be sound implementation reasons for different internal representations. Many language-based editors would cause the paste operation to fail, and a usability challenge for those systems is how to either (a) explain the failure to the user, or (b) perform grammatical transformations based on what the user probably meant to do so that the paste operation can succeed [46,73]. Here again, some of the problems can be patched, but the fundamental weakness remains: exposing in the user interface too much of the irrelevant (to users) underlying technology.

2.2.4 Views

Usability Problem 4 Monolithic document presentations that fail to exploit all the information available in the programming environment.

As argued above, the true structure of software from the perspective of experienced professionals is complex and multifaceted. 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* [75,120] and *clichés* [106] are deep) to be useful.

In some cases, many kinds of *meta-information* can be superimposed on the rich text display of the document, for example by choice of font, coloration, and background shading. In other cases, reorganization and filtering of the information is more appropriate. The table of contents and index are examples of this approach for natural language documents; they 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. This leads to the next requirement for language-based editing systems, one that helps assure that users will derive maximal benefit from the rich sources of information present in the environment.

Design Requirement 4 Multiple, alternate views.

2.2.5 Service Degradation

Usability Problem 5 Brittle system behavior in the presence of ill-formed programs or inconsistent system information.

A persistent and general problem with language-based tools is that they fail to degrade gracefully in the presence of malformed, incomplete, or inconsistent information. To a user, however, these are the natural states for documents being managed—designers of tools should make every effort to continue supporting the user when documents are in these states. This observation leads to one of the most technically challenging requirements for a language-based editing system.

Design Requirement 5 Uninterrupted service.

Ill-Formedness

Inherent in the syntax-recognizing approach is acknowledgement that documents being modified are more often than not **ill-formed**: at variance with an underlying language definition. Ill-formed documents are often said to contain "errors," a pejorative term reflecting the limitations of many analysis methods. Many language-based editors that permit text-oriented editing inherit this bias. Unable to analyze ill-formed documents, these systems insist that the user correct any newly introduced "errors" before proceeding. Often justified as a service, because it limits the extent and duration of "errors," 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. An "error" may often 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 [76].

Incompleteness

Incomplete documents represent a special case of ill-formed ones, but they correspond to natural intermediate states for documents being constructed by users and can potentially be treated more appropriately by language-based editing systems.

This challenge has been well addressed by technology developed early in the history of language-based editing systems. The first systems were so restrictive that users could only construct documents by invoking grammar rules starting from the goal nonterminal in the language, implicitly building a derivation of the desired program. The standard technique became known as the "placeholder," a visible glyph that appeared in a place corresponding to an unexpanded nonterminal in the derivation tree, usually the name of the nonterminal displayed in some way to distinguish it from terminal symbols.

The standard use of placeholders in syntax-directed editing is a very restrictive paradigm. Even in hybrid systems, placeholders are not editable by the same rules as the text.

Inconsistency

Any situation where one kind of information is derived from another invites **inconsistency** between the two. The syntax-recognizing approach, where language-based information may be derived from text, is no exception. This results from design decisions to allow full text editing and not to perform analysis with every keystroke.

Inconsistency between text and derived information should not be confused with the related but separate issue of ill-formedness. Many language-based editors that permit text-oriented editing are able to resolve inconsistency only for well-formed documents. A user of such a system who has edited a document textually is not permitted to edit elsewhere in the document until those changes meet the system's requirements for syntactic well-formedness.

Inconsistency confronts the designer of language-based editing systems with a dilemma. Until the next analysis nearly all of the derived information (including diagnostics concerning ill-formedness) produced by analyzing the text is untrustworthy. On the other hand, not to support any language-based services in this state is a needless interruption of service, since a great deal of that information will continue to be correct most of the time.

2.2.6 Multi-language Support

Usability Problem 6 Awkward (or missing) mechanisms for incorporating new languages.

Software developers typically use several formal languages daily: design languages, specification languages, structured-documentation languages, programming languages, and small languages for scripts, schemas, mail messages, and the like. The construction of "little languages" is a respected programming technique [14], and these sometimes appear embedded in other languages. For example, many subroutine packages such as libraries for developing window-based applications effectively define mini-languages that determine how subroutine calls, and in particular long sequences of arguments, must be used.

Many language-based systems are built for a single language and cannot be easily adapted. Many that can be adapted still produce systems that can only handle one language at a time. Users cannot afford the time it takes to switch tools and change interfaces along with every shift in language: a language-based system must support all these different languages as smoothly and uniformly as possible. This accounts in part for the success of GNU EMACS, which has only the shallowest of language-based information available, but which can be extended and adapted easily for new languages.

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. ... However, even with a good language specification language, we believe that the definition of abstract syntaxes and finely tuned pretty-printers are non-trivial tasks that require much care and experience, especially if the language has been originally defined in terms of a concrete syntax [71].

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.

Design Requirement 6 Description-driven support for multiple languages.

Since the services that the system can provide are based on the data it derives from a document, a language *description* may include elements beyond the scope of the basic *definition* of a language. Thus, someone adding a new service to the system may need to extend some existing language descriptions to derive new data or maintain new annotations.

Finally, this issue presents a challenging design tradeoff between customization and adaptation for specific languages on one hand, and on uniformity of user interaction across languages on the other.

2.2.7 Inflexibility

Usability Problem 7 Inflexibility because of closed data models and weak extension facilities.

A usable language-based editing system must be flexible in many ways. As with most interactive tools, it must be customizable and extensible to accommodate the enormous variations among individual users, among projects (group behavior), and among sites. [71, 122].

An effective system must also be built on a flexible framework designed to accommodate many kinds of variation and evolution [51,71]. Programmability and an open architecture are essential here, permitting extension beyond that permitted by simple parameterization of existing services. The final, requirement for language-based editing systems suggests that as many degrees of freedom as possible be supported.

Design Requirement 7 Extensibility and customizability.

As an interface to tools in the environment, a language-based editing system must be capable of using a variety of information, derived by many different tools. Users opportunistically exploit many forms of information to help them understand and modify complex documents [74]. "Information gathering" is the primary task associated with important activities like program maintenance [56].

For example, many language-based systems check that a document is well-formed. The same analysis can enable a user both to edit the document in terms of its underlying language and to locate document components that violate restrictions in the formal language definition. Other kinds of interaction may require more elaborate analysis. For example, language-based formatting (sometimes called prettyprinting), traditionally based only on surface syntax, should exploit information about scopes, types, local usage, or even distinctions such as "main-line" versus "error-handling" code. Some kinds of interaction should be driven by profile information, by information from test generators, and any other tools in the environment that can produce potentially useful metainformation.

Chapter 3

The Pan System

The requirements derived in Chapter 2 are addressed by the design of *Pan*, a multi-window, mouse-based editing and browsing system that was developed to support ongoing research into integrated document development environments [12,132]. *Pan I* Version 4.0[1] [31], the currently implemented version of *Pan*, runs on SPARCstations[2] under SunOS and on DECstations[3] under ULTRIX, using the X window system [109]. Throughout this dissertation the term "*Pan*" refers to *Pan I*.

Pan I is the product of collaborative research. Two analyzers, *Ladle* [22] and *Colander* [9], provide the language-based information that drives *Pan*'s user-oriented services. Section 4.1 describes these analyzers briefly and discusses how they are integrated with the rest of *Pan*. A number of projects, some ongoing, have successfully used *Pan* as a development platform for research topics such as advanced textual presentations of programs [16], graphical presentation of data structures [98], and dynamic compilation by attributed transformational code generation [18].

This chapter introduces the *Pan* system, beginning with design requirements described in Chapter 2. A preliminary glimpse of the system itself follows, showing how *Pan* appears to its users when in operation. The remainder of the chapter discusses the approaches developed to guide its design and implementation.

3.1 System Requirements

Pan's design was motivated by observations about the nature of programming and the people who do it, and by drawing on the experience of previous generations of experimental systems with related goals. The design requirements listed in Figure 3.1 summarize the issues discussed in Chapter 2.

These requirements affect *Pan*'s implementation at every level, including the language-based analyzers that are otherwise hidden from view. One result of this research consists of insight into problems associated with the transfer of language-based technology from its batch-oriented and compiler-inspired origins into interactive editing and browsing systems. User-centered requirements such as these impose demands on the technology that diverge

[1]The version 4.0 architecture described here differs substantially from earlier *Pan* versions [9,13].

[2]SPARCstation and SunOS are registered trademarks of Sun Microsystems, Inc.

[3]DECstation and ULTRIX are registered trademarks of Digital Equipment Corp.

1. **Familiar, unrestricted text editing.**
2. **Rich information display.**
3. **Coherent user interaction with system and programs.**
4. **Multiple, alternate views.**
5. **Uninterrupted service.**
6. **Description-driven support for multiple languages.**
7. **Extensibility and customizability.**

Figure 3.1: Summary of Design Requirements for *Pan*

markedly from those imposed by traditional compilers. Section 6.1 returns to these issues.

3.2 Using *Pan*

Figure 3.2 shows a sample text editing session. On the surface *Pan* appears to users as a convenient bit-mapped, mouse-based, multiple window text editor. In the figure, two windows are open onto a text document being viewed, and a help window displays the keybinding configuration in that view.

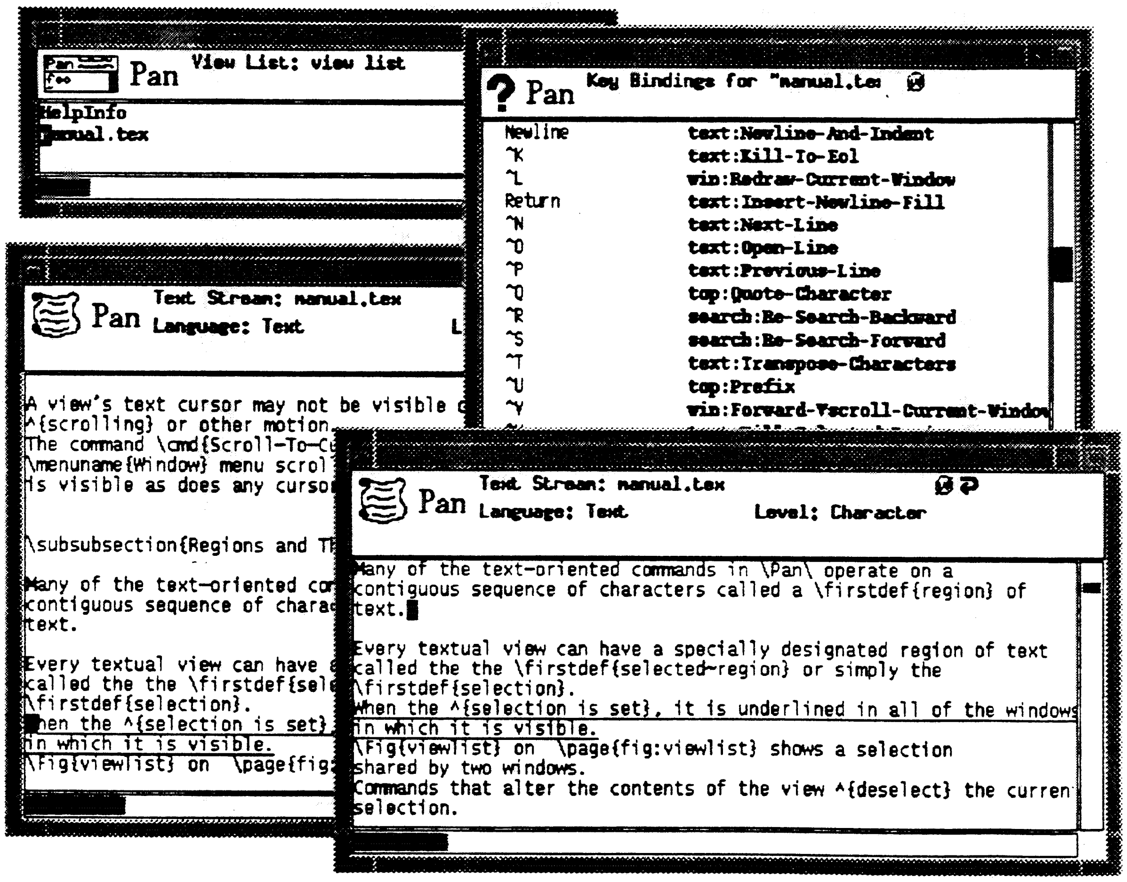
Text editing services based on familiar models encourage smooth integration into existing working environments. Users familiar with EMACS [122] find the transition between the two editors smoothed by comparable text services and compatible keybindings [31]. Generalized unlimited undo, kill-rings, text filling, character classes, customization, extension, and self-documentation are among *Pan*'s many standard services. The same text-oriented services are provided for every document, whether or not its view style includes language-based configuration. In contrast to many syntax-directed editors, one might use *Pan* for editing text without ever giving a thought to its other capabilities.

But at any time one may choose to broaden the dialogue with *Pan* and to exploit information (maintained by *Pan*) *about* the document.[4] *Pan* can be directed to use this information to guide editing actions, to configure and selectively highlight the textual display, to present answers to queries, and more.

Textual and language-based operations may be mixed freely, as shown in Figures 3.3 and 3.4. In Figure 3.3 the user has just typed in a line of new text (beginning with "`Fact :=`") by setting the text cursor with the mouse and typing. Note that the new line of text looks a bit different from the rest of the program, revealing that this text has not been analyzed and that *Pan* therefore does not yet have any language-based information about it. The user then invokes a series of language-based navigation commands that move the cursor forward a statement at a time[5] until the desired one is reached. As Figure 3.4

[4]In terms of Ballance's taxonomy [9] discussed in Section 2.1.2, *Pan* is a **syntax-recognizing** editing system.

[5]The command to move forward structurally is generic. The specific kind of structure for the command is selected from a menu, or by keystroke sequence, and appears in the panel as the "`Level`". The level



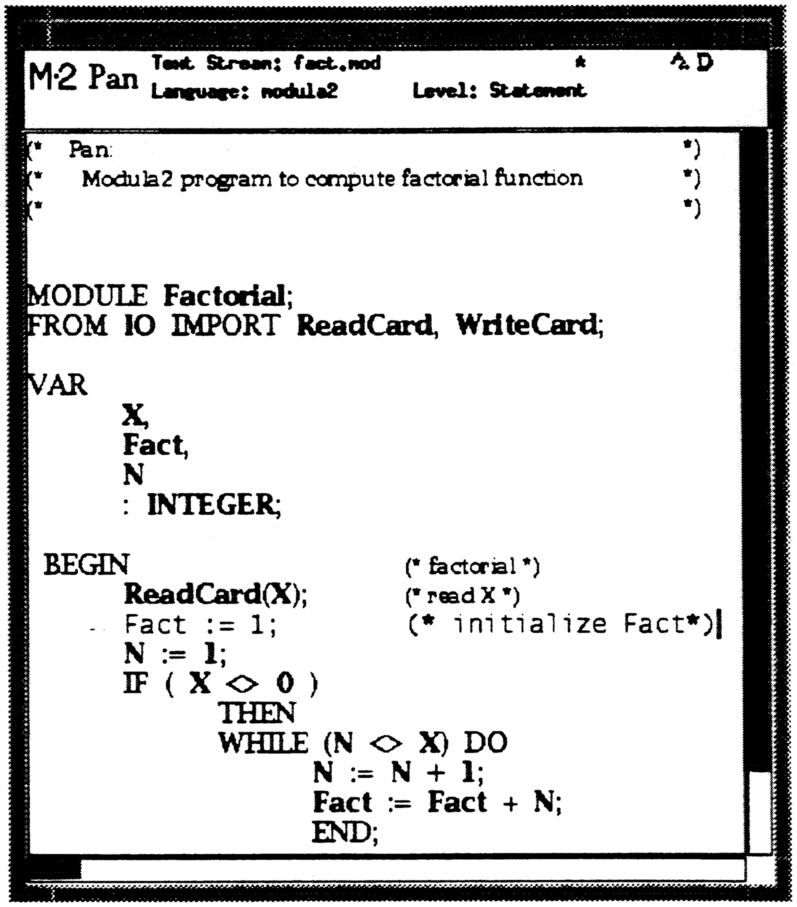
```
Pan  View List: view list
HelpInfo
manual.tex
```

```
Pan  Key Bindings for "manual.te
Newline      text:Newline-And-Indent
^K           text:Kill-To-Eol
^L           win:Redraw-Current-Window
Return       text:Insert-Newline-Fill
^N           text:Next-Line
^O           text:Open-Line
^P           text:Previous-Line
^Q           top:Quote-Character
^R           search:Re-Search-Backward
^S           search:Re-Search-Forward
^T           text:Transpose-Characters
^U           top:Prefix
^V           win:Forward-Vscroll-Current-Windo
```

```
Pan  Text Stream: manual.tex
     Language: Text              L

A view's text cursor may not be visible
^{scrolling} or other motion.
The command \cmd{Scroll-To-C
\menuname{Window} menu scrol
is visible as does any curso

\subsubsection{Regions and T

Many of the text-oriented co
contiguous sequence of chara
text.

Every textual view can have
called the the \firstdef{sel
\firstdef{selection}.
When the ^{selection is set}
in which it is visible.
\Fig{viewlist} on  \page{fig
```

```
Pan  Text Stream: manual.tex
     Language: Text              Level: Character

Many of the text-oriented commands in \Pan\ operate on a
contiguous sequence of characters called a \firstdef{region} of
text.

Every textual view can have a specially designated region of text
called the the \firstdef{selected~region} or simply the
\firstdef{selection}.
When the ^{selection is set}, it is underlined in all of the windows
in which it is visible.
\Fig{viewlist} on  \page{fig:viewlist} shows a selection
shared by two windows.
Commands that alter the contents of the view ^{deselect} the curren
selection.
```

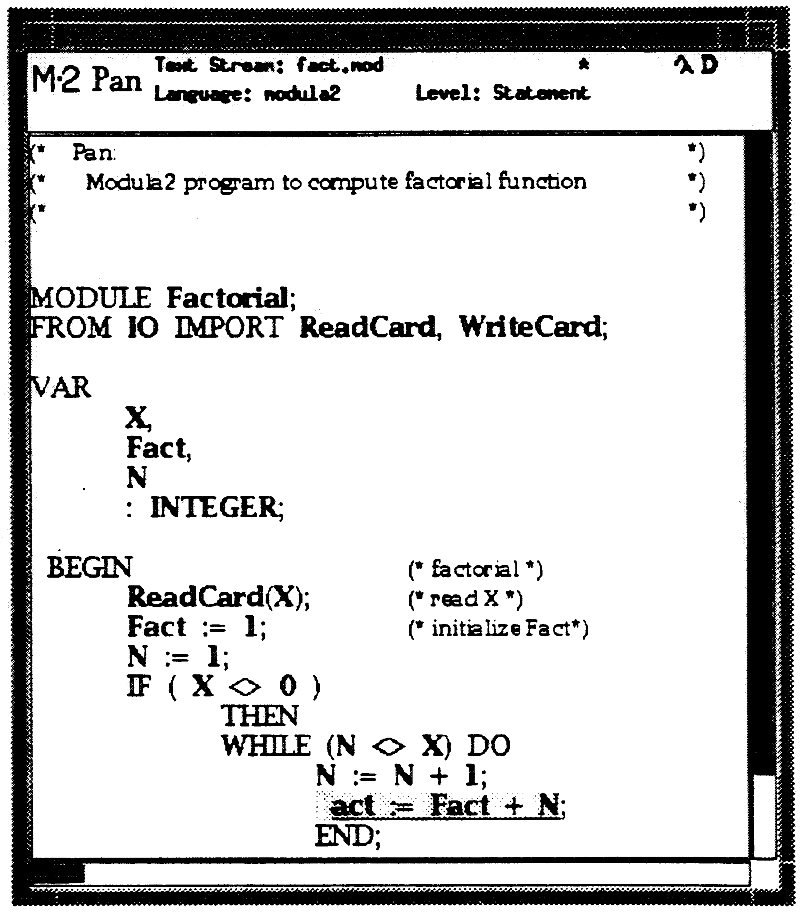
Figure 3.2: A *Pan* Session



M·2 Pan Text Stream: fact.mod * A D
Language: modula2 Level: Statement

```
(*  Pan:                                                *)
(*    Modula2 program to compute factorial function     *)
(*                                                      *)


MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard;

VAR
      X,
      Fact,
      N
      : INTEGER;

 BEGIN                      (* factorial *)
      ReadCard(X);          (* read X *)
      Fact := 1;            (* initialize Fact*)
      N := 1;
      IF ( X <> 0 )
            THEN
            WHILE (N <> X) DO
                  N := N + 1;
                  Fact := Fact + N;
                  END;
```

Figure 3.3: Using *Pan*: Entering Text



```
(*   Pan:                                                    *)
(*     Modula2 program to compute factorial function         *)
(*                                                           *)


MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard;

VAR
        X,
        Fact,
        N
        : INTEGER;

 BEGIN                          (* factorial *)
        ReadCard(X);            (* read X *)
        Fact := 1;              (* initialize Fact*)
        N := 1;
        IF ( X <> 0 )
                THEN
                WHILE (N <> X) DO
                        N := N + 1;
                         act := Fact + N;
                        END;
```

Figure 3.4: Using *Pan*: Structural Navigation

shows, all recently entered text has been analyzed as a side-effect of the language-based commands (analysis can never cause a command to fail, nor can it place the user in any special modes), and the cursor now rests at a statement — both structurally, revealed by shading the entire statement with a pale blue background, and textually, as shown by the usual inverted box. The user may now (as always) move forward either structurally or textually (probably easiest) to the erroneous operator "+", delete the character textually, type in the correct operator "*", and proceed with other tasks.

Pan can be configured to make available many kinds of information *about* documents, depending on the task at hand, using a variety of techniques. For example, Figures 3.5 and 3.6 show some of these techniques applied to language diagnostics. In Figure 3.5 a **panel flag** in the form of a red exclamation point appears near the upper right corner of the window. This particular flag reveals that some number of language errors are known to the analyzer; in some *Pan* configurations this will be the only clue available that the program is not well-formed. In this configuration, however, a **highlighter** for language errors is also available and has been toggled on by the user. This highlighter has been configured to render malformed code with red ink, and three such lines appear in Figure 3.5 (including one under the structural cursor). Highlighting draws attention to error sites, and is often all the notification that an expert programmer needs. Furthermore, the user may navigate to error sites by setting the level appropriately ("`Language Error`" is configured to be a component of the level menu) and invoking structural navigation commands (forward, backward, mouse click, and the like). Figure 3.5 show the result of such a navigation command, with the cursor landing on the "`IMPORT`" statement. Navigation helps locate error sites that may be scrolled off screen, and as a useful side effect it announces in the panel the diagnostic at each site reached. The announcement is context sensitive, however, and navigation to the same site at the "`Statement`" level would produce no such message. Finally, an alternate view may be requested that summarizes a particular kind of information, language errors in this case, as shown in Figure 3.6. Structural navigation is shared among all views, so setting the structure cursor on the line "`malformed expression`" in the language error view also causes the cursor in the primary view to be set and the window scrolled appropriately.

Chapter 4 describes in considerably more detail how *Pan*'s language-based mechanisms are constructed, and Chapter 5 discusses how they can be applied to a wide variety of useful tasks for users.

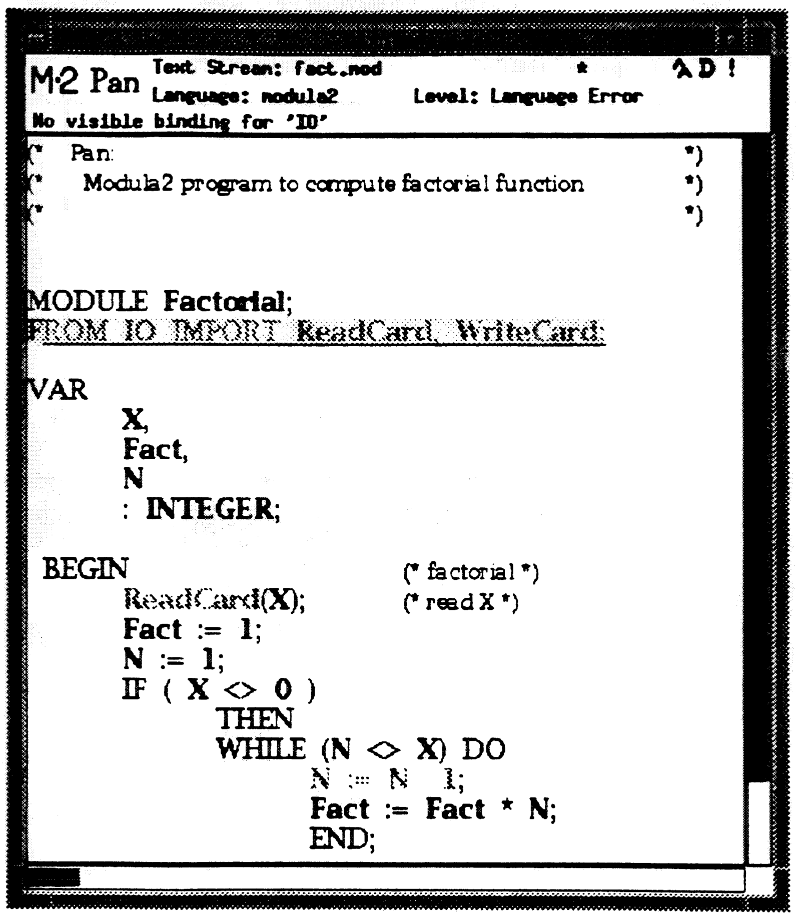
3.3 Design Principles

This section introduces the general design principles that guided work on *Pan*. All are rooted in the observation that a successful system addressing *Pan*'s goals must reflect good user interface design on two fronts.

1. The system itself must strike users as familiar, simple, and easily learned in productive increments.

2. The entire system must be considered an interface between users and documents, helping to make those documents comprehensible.

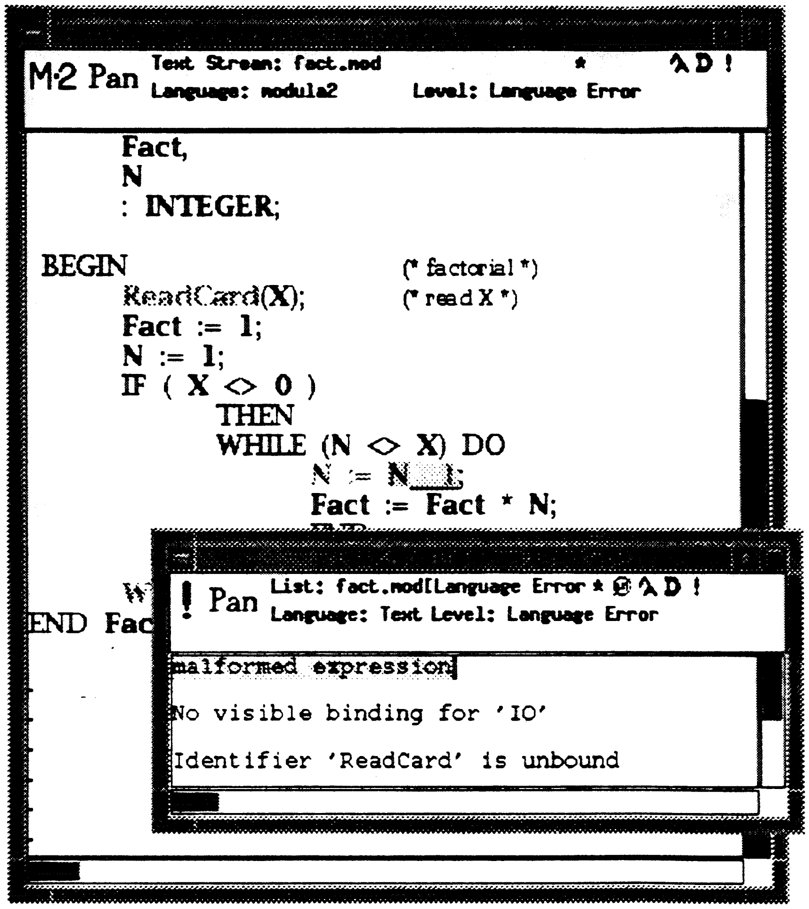
is a weak input mode that controls generic language-based commands, but does not interfere with textual commands.



```
Text Stream: fact.mod                    *
Language: modula2        Level: Language Error
No visible binding for 'IO'
```

```
(*  Pan:                                          *)
(*   Modula2 program to compute factorial function *)
(*                                                *)


MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard;

VAR
        X,
        Fact,
        N
        : INTEGER;

 BEGIN                          (* factorial *)
        ReadCard(X);            (* read X *)
        Fact := 1;
        N := 1;
        IF ( X <> 0 )
                THEN
                WHILE (N <> X) DO
                        N := N  1;
                        Fact := Fact * N;
                        END;
```

Figure 3.5: Using *Pan*: Highlighting and Finding Errors



```
        Fact,
        N
        : INTEGER;

 BEGIN                          (* factorial *)
        ReadCard(X);            (* read X *)
        Fact := 1;
        N := 1;
        IF ( X <> 0 )
             THEN
             WHILE (N <> X) DO
                   N := N  1;
                   Fact := Fact * N;
                   END
        W
END Fac
```

```
malformed expression
No visible binding for 'IO'
Identifier 'ReadCard' is unbound
```

Figure 3.6: Using *Pan*: an Auxiliary View

Design of the first sort is increasingly acknowledged as simply good practice, although very difficult for non-trivial systems. Design of the second sort has been largely ignored by previous work on language-based editing systems. The approach adopted for *Pan* is best captured by the maxim "simple system, complex documents." Stated differently, complexity of user interaction should appear to arise only from the complexity of the documents or from tasks being performed, not from the system itself.

One approach to the need for an apparently simple system is to hide any complexity associated with language-based analyzers that does not correspond to relevant complexity in programs. This is an argument for "decoupling" user interaction in language-based editing systems from the linguistic and implementation details of their enabling technologies.

Another approach is to keep system features for language-based interaction (these are called *basic services* in *Pan*'s design model) as few in number and as widely applicable as possible. The complexity behind these services, as perceived from users, should derive from the information that drives them (complexity inherent in the nature of software), not from their basic function.

The inherently complex nature of software, on the other hand, resists general approaches to simplification. Comprehension by programmers is extremely dependent on context: language, individual skills, particular software systems, and the task of the moment. Design for interaction between people and programs must include a configuration component that can only be realized in context and can therefore only be carried out by users or their close associates.

3.4 Design Layers

One realization of these general principles is an architectural framework, shown in Figure 3.7, that articulates four separate kinds of *design* that play a role in meeting *Pan*'s goals. Core language-based technology is at the lowest level, completely hidden from users, and the design of *view styles* (the *Pan* term for specialized editing contexts) is at the highest and most visible level.

This model ignores other useful decompositions of the *Pan* system in order to address the fundamental question being explored by this research: how can the power of language-based technology be exploited effectively in the context of a system built around the design requirements summarized in Figure 3.1? Weakening any of *Pan*'s design requirements would permit a simpler, less general architecture, as exemplified by systems reviewed in Chapter 2. This section discusses the motivation for each layer in the model, and describes in general terms how they interact.

Infrastructure

Many earlier language-based systems coupled their visible functionality too closely to underlying language-based representations and analysis mechanisms. *Pan*'s layered model addresses this problem by placing language-based technology in the implementation layer farthest removed from user visibility. This decomposition permits support for user models (of both the system and programs) that are largely independent of the technological particulars. For example, *Pan* view styles can be configured so that the distinction between

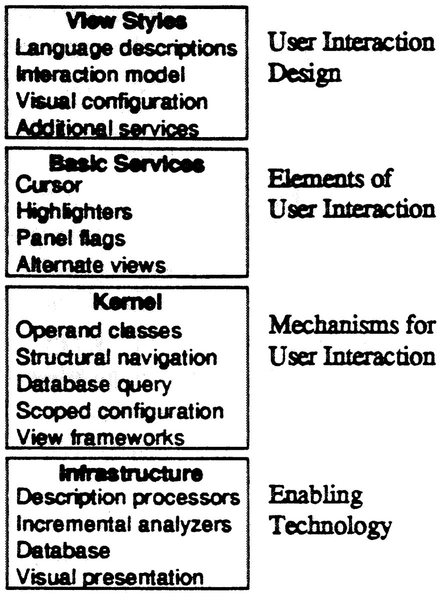


Figure 3.7: Design Layers

syntactic errors and contextual constraints (type violations, for example) is largely hidden, even though the two kinds of errors arise from separate incremental analyzers, which maintain separate data structures and use separate parts of the language description.

The assigned role of the language-based technology in this model is to gather and maintain information *about* programs, either by deriving it from the program text, by importing it, or as a side-effect of the user's interaction with higher level mechanisms. Although the technical challenges to design in this layer are great, the particular needs or shortcomings of this technology should never be permitted to place demands on users or restrictions upon their use of the system. Section 4.1 describes the language-based machinery at this level in the *Pan* system.

Pan's facilities for text-based viewing and editing are also properly part of the infrastructure (and all other layers), but otherwise receive little mention. Likewise neglected in this discussion will be the machinery associated with window-system interaction, event capture and dispatch, use of widgets, and a host of other details.

Kernel

The kernel layer, described in more detail by Section 4.2, addresses three needs. The first is to decouple the rest of the system from the representations and analysis mechanisms of the language-based infrastructure. This layer provides low-level mechanisms that embody a more abstract and general view of language-based data than is provided by the analyzers themselves. This view is provided for the implementation of generic language-based mechanisms and is not exposed directly to users. The presence of this layer makes it possible to replace *Pan*'s analyzers and data representations without affecting the system above the kernel layer; this has already been done successfully in one *Pan*-supported research project [18] and another is in progress.

The second need is for uniform user interaction across multiple languages. All the mechanisms in this layer (and by extension all the services built upon them) are language-independent, relying on minimal language-specific glue as part of each view style configuration.

The third need is for extensible user services. The low-level mechanisms are simple and general enough that additional services can often be prototyped and migrated into the system with no additional requirements at this level.

Basic Services

The third layer in *Pan*'s model implements mechanisms that the user sees and possibly invokes directly. These contribute basic functionality from the user's perspective, and thus implicitly constitute a conceptual model of the system. Section 4.3 describes the elements of this model.

It is crucial that this conceptual model be as simple and unobtrusive as possible, that it be modeled on familiar concepts, and that it be uniform across languages. A successful conceptual model would blend imperceptibly into the framework of text editing, and would thus disappear entirely from the user's explicit awareness.

Unfortunately no intrinsic *property* of the system can satisfy a criterion so intimately bound to human endeavor. It is instead the *relationships* that matter: among the system, users, their tasks, and their languages. Thus, *Pan*'s basic functionality must be complemented by user interface design in the form of careful configuration, the subject of the next layer.

View Styles

All user interaction in *Pan* takes place in the context of a *view*, and every view is an instance of some *view style*. A user might appropriately think of the entire system as a extensible collection of closely related view styles. A view style is similar to a separate editor in the sense of editor-generating systems [50,65,105]. Unlike separately generated editors, *Pan* view styles share dynamic configuration data from which each view style inherits much of its behavior, they exhibit highly uniform behavior even when different languages are involved, and they share other run-time data. A view style is also somewhat analogous to a GNU EMACS *mode* [121], but with much richer structure and an explicit shared representation independent of its instances. For example, a simple familiy of view styles might contain one per language, presenting similar interfaces and analogous services in each.

From the implementor's perspective, view styles are the locus of configuration. No service is delivered to the user without the bindings (for example, between keystroke/mouse-button events and system actions) defined by a view style. The default behavior of the system, independent of any language-based view styles, is configured completely by specifications for default view styles. A language-based view style contains directives to *Pan*'s analyzers (Section 4.1) appropriate to a particular underlying language, but view styles and languages are not the same thing; in particular, many view styles might be based usefully on a single language.

To the view style *designer*, who plays an essential role in meeting *Pan*'s user-centered goals, each view style is the specification for a particular user interface. A view style is a piece of user interface design that takes into account not only an underlying language, with its unique aspects, but also the intended users and the tasks at hand. For example a family of view styles might be use a single underlying language but each might be adapted for a particular task: language learning, small program construction, program construction as part of a large system, code reengineering. Likewise, a family of view styles might be designed for all the languages used by a particular group carrying out a particular task. In both cases, the view styles must promote uniform, predictable modes of interactions, with specialization only as needed. Section 4.4 discusses view styles and the role of the designer in more detail.

3.5 Design Metaphors

A system model must, in addition to meeting functional requirements, present users with a coherent whole. One approach to ensuring this coherence is the application of a few simply stated **design metaphors** to all elements of the model. Design metaphors are seldom explicit in user interaction with the system, but successful ones are self-evident. They correspond to the kind of explanation an experienced user might offer to a less experienced one, when giving an informal overview of the system. The following metaphors have guided the design of *Pan*'s basic services.

Design Metaphor 1 Augmented Text Editor

Pan is a text editor whose text-based services are always available in every context. All other services are additions to the text editor; some may be used to guide text editing but they never interfere. A useful analogy for these other services would be with spelling checkers in text editing systems.

Design Metaphor 2 Heads-Up Display

Many of *Pan*'s services present information *about* a document as enhancements to the text display. Designed to exploit the enormous potential bandwidth of human vision, these enhancements never interfere with standard text-based services. This approach is analogous to "heads-up" displays in which data are displayed by superposition onto the users primary visual field. These are considered especially effective aircraft pilots, for example, allowing pilots to attend continuously to the most important part of their job: looking and flying. Here the primary visual field is considered to be a display of program source, from which the user should be distracted as little as possible.

Design Metaphor 3 Imperfect World

Although *Pan* exploits knowledge of underlying languages, it operates no more differently in the presence of "language errors"[6] than does a text editor in the presence of spelling errors.

[6]Even the term "error" is inappropriate, as argued in Section 5.4, but the usage is well established.

Design Metaphor 4 Smart versus Dumb Services

Some of *Pan*'s language-based services appear not as distinct mechanisms, but as generalization of familiar, text-based services. A generalized service typically changes character dynamically: **dumb** when only 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.

Design Metaphor 5 Strict versus Gracious Services

Many of *Pan*'s 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 their 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.

Chapter 4

Layers of Design in Pan

This chapter describes architectural features of *Pan* developed to address the goals of this research. This description presents a structural view of the system, with emphasis on how the system is constructed and how it works. Chapter 5 presents a complementary task-oriented view that emphasizes what the system does and how it can help its intended users.

The presentation in this chapter follows the layered model of design developed for *Pan*, introduced in Section 3.4 and depicted in Figure 3.7 on page 30. One useful characterization of those layers involves *visibility*:

- The lowest layer consists of enabling technologies in the form of language-based analyzers and support for window-oriented interaction. Although languages and windows are of direct concern to users, the implementation details at this level are largely hidden from users and from the rest of the system

- Mechanisms implemented by the **kernel** (Section 4.2), the second layer from the bottom, support user interaction but are not directly visible to users.

- **Basic mechanisms** (Section 4.3) on the other hand, correspond to system features that the user does see. These contribute, along with conventional text editing, to the user's model of the system.

- From basic mechanisms, combined with *Ladle* and *Colander* descriptions of underlying languages, are constructed **view styles** (Section 4.4), the uppermost layer in the model. Each view style constitutes a complete editing context and is designed to support a particular class of users performing a particular task using a particular underlying language. Implicit in this design is a user model of document structure appropriate to the context.

Chapter 5 revisits many of the same topics, but from the point of view of users and organized by ways in which mechanisms are *used*.

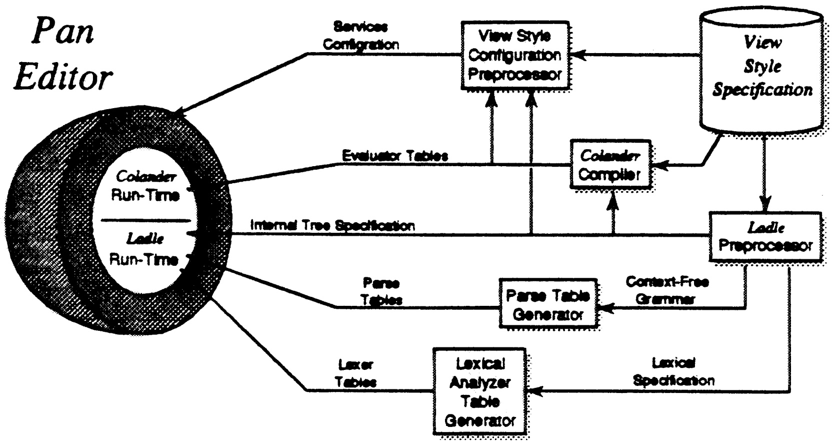


Figure 4.1: Preprocessing Language-Based View Style Specifications

4.1 Language-Based Infrastructure

Program analysis in *Pan* relies on two parts of the infrastructure developed in the course of closely related research: *Ladle*[1] [22], and *Colander*[2] [9]. This section describes each in more detail and discusses how the two cooperate and make information available to the rest of *Pan*.

4.1.1 Language Description Processing

A language-based view style contains information for use by *Pan*'s two analyzers,[3] written in a declarative language suitable for each. *Ladle* manages incremental lexical and syntactic analysis; it includes both an offline preprocessor that generates language-specific tables and a run-time analyzer that revises *Pan*'s internal tree-structured representation to reflect textual changes.

Colander manages the specification and incremental checking of context-sensitive constraints, including, but not limited to, the static semantics of the language. A *Colander* specification may also direct that certain data derived during checking be stored and made available for general use. Like *Ladle*, *Colander* includes both an offline preprocessor and a run-time analyzer. Figure 4.1 illustrates the flow of information from language description to the run-time *Pan* system, where the preprocessed data may be either preloaded or loaded on demand at run time.

4.1.2 Ladle

An abstract syntax is described to *Ladle* using an augmented context-free grammar, which also specifies the tree-structured internal representation. The internal representation im-

[1]Language Description Language

[2]Constraint Language and Interpreter

[3]In the current implementation, each document must be composed using a single language. *Pan*'s architecture and algorithms support documents composed from multiple languages, but the current implementation does not.

plicitly defines the universe of structural program components accessible to *Pan*'s language-based services.

Additional information enables *Ladle* to convert textual representations to tree-structured representations and vice versa:

- The lexical description may include both regular expressions and bracketed regular expressions, that is, expressions with paired delimiters such as quote marks. Bracketing can be either nested or simple.

- The grammar for the abstract syntax is augmented by specifying those productions necessary to disambiguate the original (abstract) grammar or to incorporate additional keywords and punctuation. *Ladle* constructs a full parsing grammar from the additional productions and the grammar for the abstract syntax.

- Optional directives tune *Ladle*'s syntactic error recovery mechanisms (invoked during parsing). These directives also have important effects on the editing interface (Sections 5.2 and 5.4).

Internally, *Ladle* manipulates two context-free grammars: one describing the abstract syntax and the other used to construct parse tables. The two must be related by **grammatical abstraction**[4] [10], a relation ensuring that:

1. The abstract syntax represents a less complex version of the concrete syntax, but structures of the abstract syntax correspond to structures of the concrete syntax in a well-defined way. Both grammars describe "almost" the same formal language, subject to the renaming or erasing of symbols.

2. Efficient incremental transformations from concrete to abstract and from abstract to concrete can be be generated automatically—no action routines or special procedures are necessary. The transformation from concrete to abstract is triggered directly by actions of the parser.

3. The transformation from concrete to abstract is reversible, so that relevant information about a concrete derivation can be recovered from its abstract representation. This property allows the system to parse modifications incrementally without having to maintain the entire parse tree.

4. The relationship between the two descriptions is declarative and statically verifiable so that developers can modify either syntax description independently. This approach allows a high degree of control over both the structure of an internal representation and the behavior of the system during parsing.

Grammatical abstraction is structural; it does not use semantic information to identify corresponding structures. Two examples of grammatical abstraction appear in Figure 4.2. The concrete grammar G_1 describes the syntax of conditional statements. The fragments $\widehat{G_1}$, and $\widehat{G'_1}$ are both allowable (but different) grammatical abstractions from the fragment G_1, assuming that the symbols $\langle stmts \rangle$ and $\langle expr \rangle$ are interesting to the grammar writer.

[4]Butcher later recast this work in terms of **grammatical expansion** [22].

$$\begin{array}{lllll}
1) & \langle \mathit{stmt} \rangle & \rightarrow & \langle \mathit{if\text{-}stmt} \rangle \text{ ";"} \\
2) & \langle \mathit{if\text{-}stmt} \rangle & \rightarrow & \langle \mathit{if\text{-}part} \rangle \langle \mathit{else\text{-}part} \rangle \\
3) & \langle \mathit{if\text{-}part} \rangle & \rightarrow & \mathsf{if}\ \langle \mathit{expr} \rangle\ \mathsf{then}\ \langle \mathit{stmts} \rangle \\
4) & \langle \mathit{else\text{-}part} \rangle & \rightarrow & \epsilon \\
5) & & | & \mathsf{else}\ \langle \mathit{stmts} \rangle
\end{array}$$

Concrete Grammar G_1

$$\begin{array}{llll}
1') & \langle \mathit{stmt} \rangle & \rightarrow & \mathsf{if}\ \langle \mathit{expr} \rangle\ \mathsf{then}\ \langle \mathit{stmts} \rangle \text{ ";"} \\
2') & & | & \mathsf{if}\ \langle \mathit{expr} \rangle\ \mathsf{then}\ \langle \mathit{stmts} \rangle\ \mathsf{else}\ \langle \mathit{stmts} \rangle \text{ ";"}
\end{array}$$

Abstract Grammar $\widehat{G_1}$

$$\begin{array}{llll}
1'') & \langle \mathit{stmt} \rangle & \rightarrow & \langle \mathit{expr} \rangle \langle \mathit{stmts} \rangle \\
2'') & & | & \langle \mathit{expr} \rangle \langle \mathit{stmts} \rangle \langle \mathit{stmts} \rangle
\end{array}$$

Abstract Grammar $\widehat{G'_1}$

Figure 4.2: Grammatical Abstraction

The *Ladle* preprocessor generates the tables needed to describe the internal tree representation as well as auxiliary tables needed during incremental parsing and error recovery. Standard lexical analyzer generators and LALR(1) parser generators are also invoked, as shown in Figure 4.1.

Ladle descriptions have been written for Ada, Modula-2, Pascal, FIDIL [54], *Colander*, and for *Ladle*'s own language description language. Descriptions are being developed for a variety of other languages, including C and C++.

4.1.3 Colander

Colander supports the description and incremental checking of contextual constraints [9]. Constraints include non-local aspects of a language definition, for example name binding rules and type consistency rules, as well as extralingual structure. Examples of the latter include site or project-specific naming conventions, design constraints, and complex non-local relationships.

Colander's approach is based on the notion of *logical constraint grammars*. In a logical constraint grammar a context-free grammar is used as a base. Contextual constraints are expressed by annotating productions in the base grammar with goals written in a logic programming language,[5] thereby associating a set of goals with each node of the tree representation maintained by *Ladle*. Program analysis is modeled as simultaneous goal satisfaction for all goals at all nodes in the tree.

Colander itself has four subcomponents: a compiler, a consistency manager, an evaluator, and a database. The *Colander* compiler generates the code used by the evaluator[6] as

[5]Logical constraint grammars should not be confused with *constraint logic programming* [25] a generalization of logic programming.

[6]The compiler actually uses *Pan* to parse language descriptions. This involution is one example of how

well as the run-time tables required for consistency maintenance. The consistency manager, a simple reason maintenance system [32,118], invokes the evaluator to (re)attempt a goal when it is either unsatisfied or some fact upon which it depends has changed. The incremental evaluator, in turn, collects the information maintained by the consistency manager, and stores it in the database for shared access by other services [11].

Colander's database is logic-oriented, but structured in useful ways. For example, all tuples reside in one of three kinds of **collections**:

Datapool: An arbitrary collection of **facts** that can be named and treated as a single unit. A typical application is to model a program scope; typical facts include local declarations and relationships to other scopes.

Entity: A collection of **properties** that, unlike facts, are single-valued. A typical application is to model particular objects such as a variables and procedures.

Subtree: A collection of properties, instantiated automatically to mirror the abstract tree representation maintained by *Ladle*. Tree properties include, but are not limited to structural relationships like "parent".

As an important special case, **maintained subtree properties** may be declared for automatic maintenance by the consistency manager (via lazy evaluation), using node-specific procedures that can mimic attribute grammar evaluation. These properties are especially useful for computing properties such as expression types. **Client properties** are also subtree properties, but may be set by outside clients of *Colander*. Consistency maintenance is available for these too, making more general kinds of computation available to clients of *Colander*. Other features include database triggers (actions invoked by pattern matching on database transactions) and diagnostic messages attached to goal failure points, and used as described in Section 5.4.

To date, logical constraint grammars have been used to define the static semantics of programming languages, including Modula-2 and Ada, to express some aspects of design semantics, and to describe and maintain prettyprinting information. Other problems that can be expressed using logical constraint grammars include the kinds of analysis performed by tools like Masterscope [82] and Microscope [1].

4.1.4 Language-Based Analysis

Program analysis in *Pan* involves sequential invocation of incremental *Ladle* and *Colander* analyzers. Under what circumstances this happens is a policy question to be discussed in Section 5.6 and again in Section 6.1 . This section describes analysis in the context of the infrastructure.

Pan's text representation includes change markers on a per node basis, where a text node corresponds initially to a line of text but may subsequently fragment during editing. Text editing primitives cooperate with the lexer by marking newly inserted text and the locations of deleted text.

Ladle processes textual changes in two phases: lexical and parsing. An incremental lexical analyzer synchronizes a stream of lexemes with an underlying text stream, updating

Pan **is used to support itself.**

only the changed portions of the lexical stream and resetting the change markers in the text stream. The lexical analyzer maintains a summary of changes for use by the incremental parser.

Ladle's incremental LALR(1) parser revises the tree-structured representation in response to lexical changes. This parser can create a tree from scratch, but in response to lexical changes it need only modify affected areas of the tree. It uses a variant of an algorithm by Jalili and Gallier [59]. During incremental parsing, the algorithm first "unzips" the internal tree along a path between the root and the leftmost changed area. The algorithm concludes by incorporating changes and "zipping up" the unzipped portion. When unzipping and zipping up, tree nodes representing subtrees are broken apart and then reconstituted. For the benefit of *Colander* and other clients, *Ladle* classifies tree nodes after each parse: newly created, deleted from tree, reconstituted, and unchanged.

Colander's incremental analyzer proceeds in three phases, the first of which records structural changes reported by *Ladle*. In this phase the analyzer removes from the database all nodes reported deleted and, using consistency maintenance, adds to the evaluator's worklist any remaining goals whose satisfaction depended on those nodes. Similarly, newly created nodes are entered into the database and all associated goals are added to the worklist. Information associated with reconstituted nodes is retained but marked for possible updating.

Colander's next two phases are based on a partitioning of the goals associated with each node: those whose primary use is to establish the context used by that structure or by its substructures, and those goals whose primary use is to express a contextual constraint. Goals in the first class are called **first-pass** goals, and are evaluated during a top-down preorder walk of the internal tree. Goals in the second class are called **second-pass** goals, and are evaluated after the first-pass goals.[7]

4.2 Kernel Mechanisms

The kernel layer in *Pan*'s framework (Figure 3.7) addresses the earlier observation that the functional behavior of language-based editing systems should not be coupled too tightly to the details of the enabling language-based technology. This layer:

- provides a powerful substrate upon which user-oriented services and extensions can be built without being excessively coupled to the functionality of language-based analyzers in the infrastructure;

- isolates the implementation of higher-level mechanisms from the implementation details of the infrastructure, for example to permit major changes to the tree representation without affecting the implementation of higher layers.

- provides mechanisms that are language-independent (or, more accurately, language configurable) so that similar functionality and a uniform style of user interaction may be provided with as little language-specific configuration "glue" as possible.

[7]In the current implementation, second-pass goals are evaluated after the first-pass goals of the subtree's children have been evaluated.

This review of the kernel layer begins with a brief mention of conventional text editing services. It then describes kernel mechanisms developed specifically for *Pan* and concludes with an overview of *Pan*'s extension language, the medium through which many of these mechanisms are made available.[8]

The low level design and implementation details presented in this section are deliberately kept from the user's view for the same reason that details the language-based technology of the infrastructure layer are: to keep implementation choices from dictating design for user interaction. Section 4.3 following describes how these kernel mechanisms actually appear to the user, and Section 4.4 shows how a designer configures these mechanisms to create view styles for interaction.

4.2.1 Text Representation and Rendering

Pan is implemented at every level as a text editor. Text-related mechanisms in the kernel are exploited both by text editing services visible to the user and by the implementations of many other language-based services.

Text Stream Representation

Pan's internal text stream representation [131] supports conventional text-oriented services. Cursor movement primitives include both stream-oriented (1-D) and grid-oriented (2-D) motion, as well as movement to marked locations. Stream oriented motion may be pattern-based (regular expression search) or character-class based (skipping to/over characters in designated categories, for example "whitespace"). Primitive editing operations operate either at the cursor or at an explicit text selection, and are completely reversible to support *Pan*'s unrestricted undo. These primitives support more complex text operations, for example search-replace, paragraph filling, and kill-rings (clipboards).

Less conventional are text **sticky pointers**; these refer persistently to individual characters, but incur little overhead during text stream modification [34]. Nearly all low-level text services use sticky pointers; more significantly, they help support complex mappings between text and lexical stream representations.

Text Rendering Effects

Transparent to *Pan*'s conventional text editing functions are eight bits of **display data** associated with each character. Higher-level services use the primitives listed in Table 4.1 to modify display data in regions of text. Font selection (up to 8 possibilities per view) is independent of ink color (4 possibilities per view). Two kinds of background shading are independent of both font and ink selection, and independent of each other. Text underlining is also independent and similarly implemented, but it is reserved by the text editor for textual selection.

The text rendering engine interprets each character's display data using context-sensitive **color maps** and a **font map**, implemented as option variables (Section 4.2.9), which specify particular colors and fonts. Fonts may be proportionally spaced and mixed in size. Color

[8] A few experimental services that are not yet language-based use many of these same mechanisms: a browsing interface to the file system; a hypertext-like browser for UNIX man pages; and an elaborate internal help and documentation system that can be configured for each category of user.

Form	*Parameter*
`Set-Font`	Font map index: `0-7`
`Set-Ink`	Foreground color map index: `0-3`
`Turn-Mode-Off`	Background color map id `:color-bg1` or `:color-bg2`
`Turn-Mode-On`	Background color map id `:color-bg1` or `:color-bg2`

Table 4.1: Text Rendering Forms in the Extension Language

maps are defined in pairs, a primary and an alternate version of each, used during periods of document consistency and inconsistency respectively.

4.2.2 Graphical Rendering

Although primarily a text editor, *Pan*'s kernel includes some support for rendering simple graphical objects. Graphical presentation is limited at present to trees with textually labeled nodes, but the organization permits the addition of more general layout algorithms. User manipulation of graphical displays is likewise not supported yet. As with *Pan*'s text rendering effects, many configurable options control drawing parameters, for example fill colors, line colors, and line thickness.

4.2.3 Analysis Control

Pan's incremental language-based analyzers, described in Section 4.1, may be run at any time. Under what circumstances this happens is a matter of policy and configuration.

Immediately after analysis the text of a document is **consistent** with the derived data (internal tree representation and database). Any subsequent textual modification to a document causes text and derived data to be **inconsistent** until another analysis has occurred. [9] *Pan*'s language-based analyzers are unable to determine or even estimate the consequences of unanalyzed textual changes, so no derived data can be guaranteed correct during periods of inconsistency.

According to the "Strict versus Gracious Services" design metaphor, a service that depends on language-based data may either be **strict**, in which case it may not be invoked in the context of an inconsistent document, or it may be **gracious**, in which case it may be invoked at any time. The implementation of a gracious service must be prepared to function in two modes: an **exact** mode when derived data is reliable and an **approximate** mode otherwise (when a document is inconsistent). A gracious service may be configured to operate either strictly or graciously, but a service whose implementation does not support approximate operation must always operate strictly. Services operating strictly are usually configured to trigger analysis automatically when invoked, ensuring that they never operate inappropriately.

[9] This will be shortened by saying that the document being viewed is either "consistent" or "inconsistent".

4.2.4 Database Access

An important aspect of *Colander*'s operation is the maintenance of a database, described in Section 4.1.3. The organization of a document's database is directed by the *Colander* description being used, and much of the data recorded in the database is intended to facilitate efficient incremental analysis.

Two kinds of client interface enable other services to exploit the database with minimal interaction between service implementation and *Colander* description. Complete independence is possible only in limited cases, since the description configures the database. *Colander*'s native database interface resembles Prolog; its paradigm for interaction and its logic-based intermediate language differ sharply from COMMON LISP, its implementation language. The following client interfaces hide the details.

Node Properties

The *Colander* database permits assignment and reading of arbitrarily named properties on nodes of the internal tree representation. This provides a convenient mechanism for externally managed tree attribution, accessible via the forms listed in Table 4.2. The interactive forms may be accessed directly by users under some circumstances. When node

`Get-Node-Property`	`Interactive-Get-Node-Property`
`Remove-Node-Property`	`Interactive-Remove-Node-Property`
`Set-Node-Property`	`Interactive-Set-Node-Property`

Table 4.2: Node Property Access Forms in the Extension Language

properties are declared in the underlying *Colander* description, clients may interact with the analyzer by setting and reading node properties that may be involved in other computations. When not declared in the underlying *Colander* representation, these **extrinsic properties** are completely transparent to the analyzer, and no goals within the *Colander* description may depend on such a property.

Generalized Queries

When more complex client access to the database is necessary, a certain amount of coordination with *Colander* descriptions is unavoidable. To make this as convenient as possible, groups of **generalized queries**, such as those listed in Table 4.3, permit access following simple models that are not language-specific; client services may use generalized queries without any language dependence. Generalized queries only work when an underlying *Colander* description defines query functions of the same name. Binding is dynamic, so that any language-based view style loaded at any time may use generalized queries. Invocation of a generalized query for which the underlying *Colander* description has no counterpart produces a run-time warning that the requested service is unsupported.

`query-node-entity`	Entity to which an identifier refers.
`query-node-scope-name`	Scope name of entity to which identifier refers.
`query-entity-declaration`	Declaration node of entity.
`query-entity-type-string`	Textual description of entity's type.
`query-entity-value`	Value of entity if known.
`query-entity-instances`	List of all uses of entity.

Table 4.3: Generalized Queries Concerning Named Entities

4.2.5 Internal Object/View Model

Conventional text editing in *Pan* is augmented by an object/view mechanism that provides familiar behavior in the ordinary case (where a user simply visits and expects to see the contents of a text file document) but which also permits the user to see simultaneous multiple views on a document, each presenting different aspects of the document.

Pan's object/view model is transitional, designed to permit experimentation with multiple views while the architecture of its successor is developed. For this reason, the model permits but does not directly support any particular object/view relationships, for example Garlan's data-sharing views [43] or the Model-View-Controller paradigm [69] developed for Smalltalk-80. Implementation restrictions (concerning the the interface between *Ladle*'s lexer and the text representation) permit only one view per object that may be modified by the user.

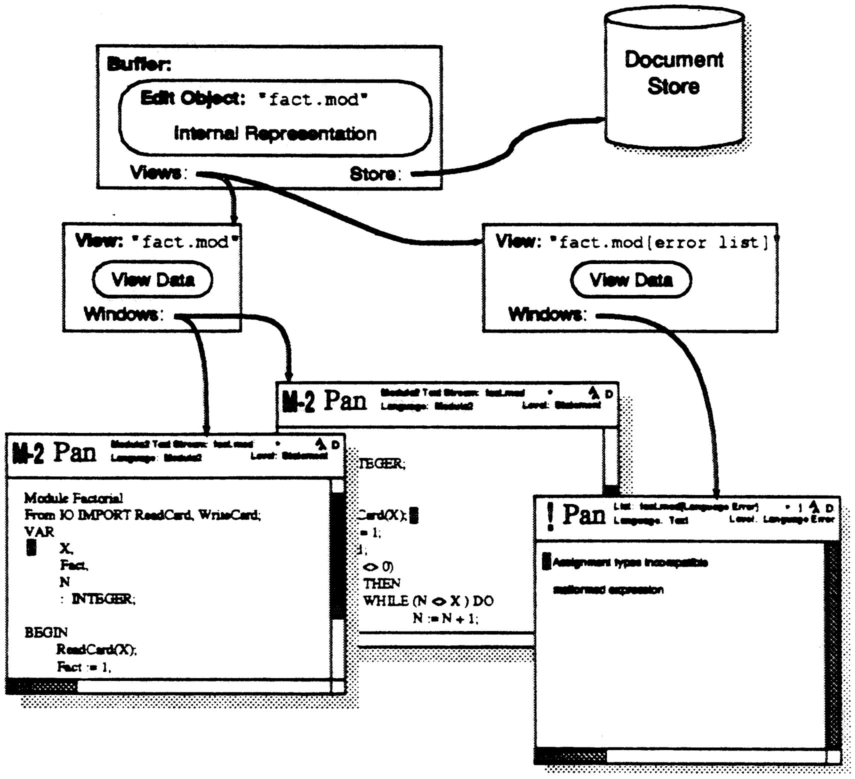
Pan's object/view model rests on four internal concepts: edit object, buffer, view, and window. Figure 4.3 depicts the relationship among these concepts for a file being edited with two views active and three windows open.[10]

Edit Objects

An **edit object**[11] is anything *Pan* users may see and edit, typically software documents. Edit objects may also be internal data structures; for example the list of views active during a *Pan* session is represented as an edit object that can be viewed and manipulated. Storable edit objects reside in the UNIX file system at present, although both design and implementation anticipate more appropriate persistent storage. Figure 4.4 shows the current implementation class hierarchy for edit objects. A typical language-based edit object has the properties listed in Table 4.4, where the properties deriving from each of its relevant parent classes (both class slots and instance slots) are indicated.

[10]The screen snapshot appearing in Figure 4.7 on page 61 shows this scenario as it appears to the user.

[11]This use of the term *object* should not be confused with the notion of object as understood in object-oriented programming, although the two are loosely related. Scofield used "emendand" in roughly the same sense as *Pan*'s "edit object" [112], but the term seems not to have caught on. Many of *Pan*'s internal structures, on the other hand, including edit objects, are *implemented* as CLOS objects in the object-oriented programming sense. Discussion of internal structures will favor the terms "instance" and "class" for CLOS implementation constructs to avoid confusion.



```
Module Factorial
From IO IMPORT ReadCard, WriteCard;
VAR
    X,
    Fact,
    N
    : INTEGER;

BEGIN
    ReadCard(X);
    Fact := 1;
```

```
TEGER;

Card(X);
= 1;
1;
<> 0)
THEN
WHILE (N <> X ) DO
        N := N + 1;
```

```
Assignment types incompatible
malformed expression
```

Figure 4.3: Example: Internal Object/View Model

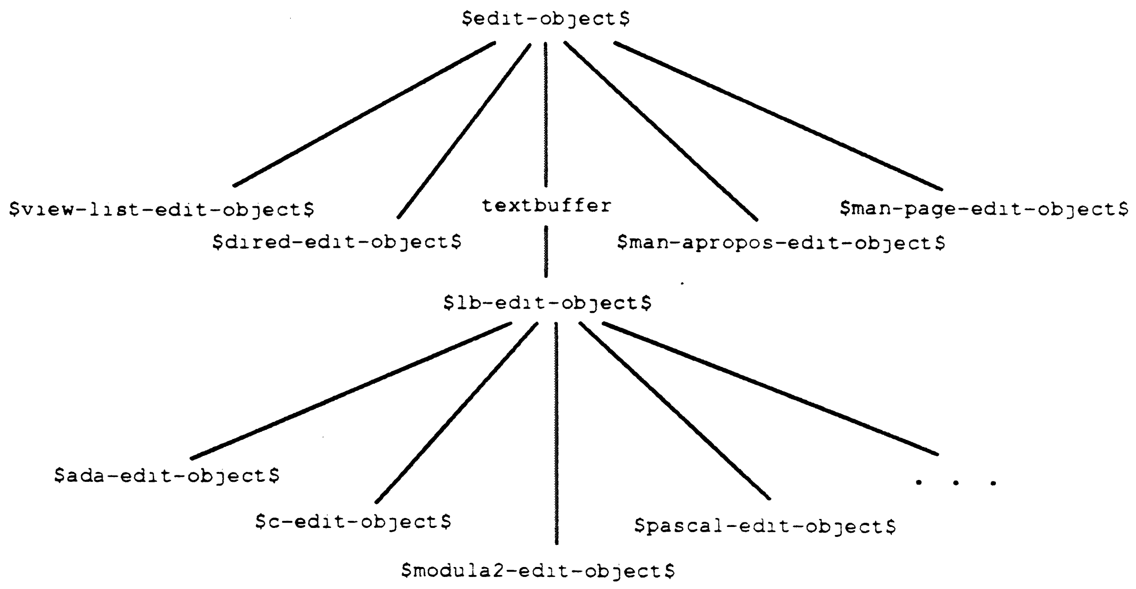


Figure 4.4: Edit Object Class Hierarchy

From parent class **$edit-object$**:

presentation name	(class) User name for class, e.g. "modula2 program".
storable?	(class) Is persistent storage possible?
default view-style	(class) Default view style for class.
protected?	User permitted to make changes?
modified?	Changes since most recent store?
undo history	List of modifications, reversible.

From parent class **$lb-edit-object$**:

language name	(class) Underlying language, eg. "modula2".
syntax data	(class) *Ladle* data for language.
semantics data	(class) *Colander* data for language.
root tnode	Root of *Ladle* tree representation.
parse current?	Has *Ladle* run since most recent changes?
local semantic data	*Colander* database for program.
database updated?	Has *Colander* run since most recent changes?
structure cursor	A distinguished node in the tree.
current query	A distinguished, named list of nodes in the tree.

Table 4.4: Properties of a Language-Based Edit Object

Buffers

A **buffer** is the container for an edit object, once created or retrieved from storage. Buffers represent two important relationships: between edit objects and persistent storage (an edit object may or may not have a storage location), and between edit objects and their associated views (a buffer may have one or more views on its edit object). Buffers are only marginally relevant to issues of visual presentation and editing, which are the business of views and windows. There is only one type of buffer, and it has the properties listed in Table 4.5.

edit object	Object being edited.
store	Persistent storage location for edit object.
backup	Storage location for edit object's backup copy.
checkpoint	Storage location for edit object's most recent checkpoint copy.
view list	Current views on edit object.

Table 4.5: Properties of a Buffer

Views

A **view** creates a *presentation* (a conceptual picture) of an edit object and may provide editing services such as navigation, selection, and modification. It is the responsibility

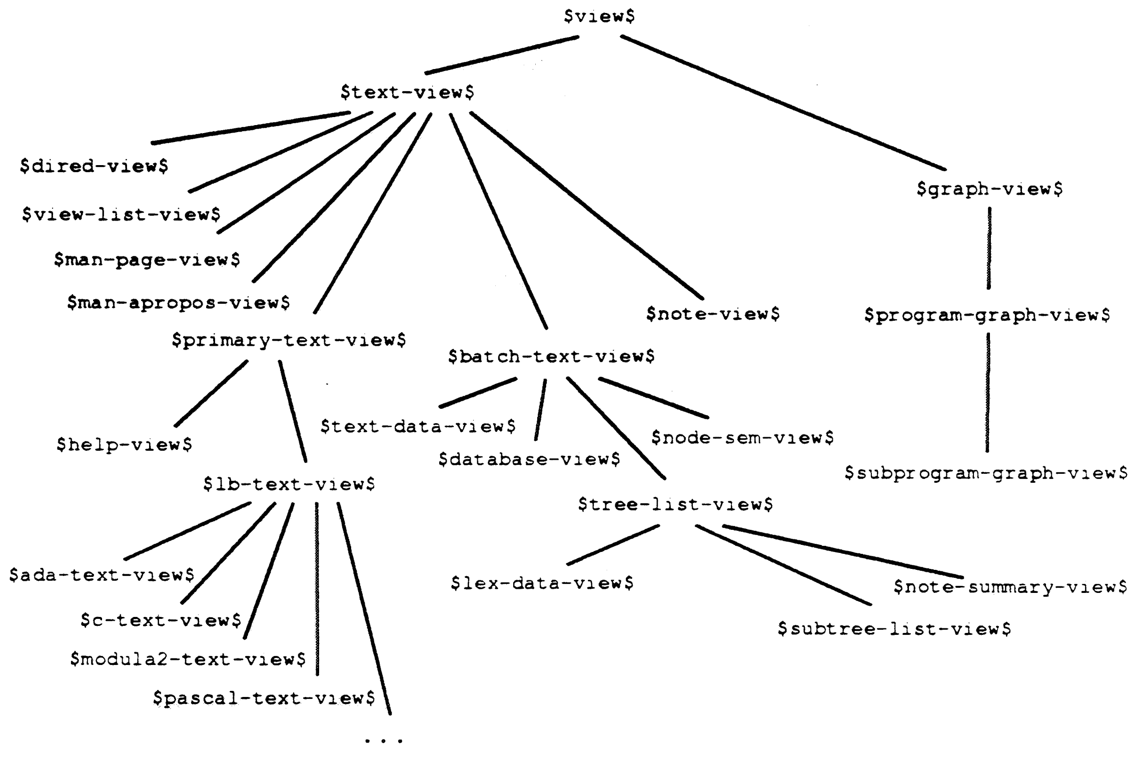


Figure 4.5: View Class Hierarchy

of a view to translate client requests for alterations into primitive operations on the edit object's internal representation. Figure 4.5 shows a subset of the current implementation class hierarchy for views. Not shown are so-called "mixin" classes that exploit multiple inheritance to add implementation features. A typical language-based, editable view has the properties listed in Table 4.6, where the properties deriving from each of its relevant parent classes are indicated. Section 4.2.8 discusses the design of selected view classes in more detail.

Windows

A **window** renders some part of a view's presentation onto the user's screen. The user chooses that part by scrolling and zooming. A window is also the locus for user interaction, from which editor context is determined, in which commands may be executed in response to user actions, and where pointing (primitive selection and other techniques) is recognized.

4.2.6 Operand Classes

Any language-based editing system must support a vocabulary of categories that services and users may use to communicate about document components. For example, a syntax-directed editor for programs may use simple syntactic categories such as "**statement**" and "**procedure**". A *Pan* **operand class** ("opclass" in the implementation) addresses this need in a configurable way. The operand class is a purely definitional entity for specifying named subsets of document components; many of the user services described in Section 4.3,

From parent class **`$view$`**:	
`presentation name`	(class) User name for class e.g. "modula2 text view"
`view-style`	Associated view-style.
`variable table`	Variables bindings for view scope.
`window list`	Currently active windows on view.
From parent class **`$text-view$`**:	
`textbuffer`	Text stream view data.
`text selection`	A distinguished region of text in stream.
From parent class **`$lb-text-view$`**:	
`highlighter list`	Enabled highlighters.

Table 4.6: Properties of a Language-Based Text View

for example navigation, highlighting, and projection, are parameterized by operand class definitions. Membership is determined dynamically, based on specifications that can refer to any information available in the infrastructure.

An operand class has the properties listed in Table 4.7. Operand class titles describe

`name`	A symbol eg. `modula2-int-expr-opclass`.
`title`	A string e.g. "`Integer Expression`".
`documentation`	A string.
`apropos`	A list of subject words.
`predicate definition`	A complex predicate on tree nodes.
`extensional definition`	A membership enumeration.

Table 4.7: Operand Class Properties

class contents in concrete, user-comprehensible terms. Operand classes may be generic (for example "`Placeholder`"), language-specific (for example "`Statement`"), or even task-specific (for example "`Expensive Type Coercion`" used while performance tuning); the distinction is invisible to clients. Generic ones are typically predefined by the system, and the rest are generated as a side effect of language-based view style specifications. Clients of the mechanism use the access functions described abstractly in Table 4.8; only the first function is strictly necessary, but the rest permit useful optimization.

The following characteristics of the operand class mechanism reflect general goals of the kernel layer and set it apart from other language-based systems that lack an equivalent layer.

- It is much more flexible than simple grammatical partitioning, for example compared to the "phyla" approach taken in Mentor [63] and in the Synthesizer Generator [103]. Specifications can exploit any kind of information and classes can overlap. Tree nodes not in any class are effectively invisible to user services. The Synthesizer Generator

`list members`	Enumerate all current class members.
`count members`	Count all current class members.
`any members?`	Are there currently any class members?
`node is member?`	Is a particular node currently in class?

Table 4.8: Access to Operand Class Membership

achieves a similar effect with the "resting place" mechanism, but only one (anonymous) class of resting places is possible.

- It separates specification from details of syntax and implementation, unlike the Synthesizer Generator's "resting place" mechanism, for example, whose specification is embedded in the unparsing scheme. Conversely, it frees the designer of the internal tree representation (expressed in the *Ladle* description) from the kind of user-motivated guidelines for AST design suggested for Mentor [29,63]. Experience with *Colander* descriptions suggests that tree representation choices are best made to facilitate analysis.

- It permits and encourages uniform behavior across languages. For example users reasonably expect to refer to "Statements" in every language that has them. A separately defined operand class per language, each titled "`Statement`", permits uniform behavior without constraining representational decisions made as part of each language's *Ladle* specification. In other cases, a single predefined operand class, for example "`Language Error`", can be used directly for many languages.

- It permits dynamic creation of new categories.

Operand Class Implementation

Access to operand class membership data would most reasonably be supported in the language-based infrastructure, using a general-purpose analyzer and database. The need for this kind of computation, however, was not anticipated in the design of *Colander* (Section 6.1.6), so the description language lacks the necessary expressive power. A prototype implementation for operand classes is at present external to the analyzer and suffers from that lack of integration.

In the general case, operand class membership is defined by a predicate on tree nodes. For example the specification in Figure 4.6 defines an operand class that contains all expressions of type integer for the demonstration language "simple." In particular, the predicate specifies nodes whose syntactic operator is "`expression`," an abstract nonterminal, and whose property "`expr-type`," produced by a database query, is the string "`integer`." Predicates are written using a simplified declarative syntax that includes the operators listed in Table 4.9, where: `<operator-name>` is the string name of an abstract non-terminal, rule, or lexeme from the underlying *Ladle* description; `<predicate>` is a COMMON LISP function of one argument, a tree node; `<variable>` is the symbol name of a scoped *Pan* variable; and `<test-spec>` is a recursively defined specification. Appendix C includes full documentation on the predicate specification language for operand classes.

```
(simple-int-expr-opclass
  :title         "Integer Expression"
  :documentation "Simple integer expressions."
  :definition    (:and (:operators "expression")
                       (:predicate
                        #'(lambda (tnode)
                            (string= (sem:Get-Node-Property
                                       tnode
                                       'expr-type)
                                     "integer")))))
```

Figure 4.6: Example Operand Class Specification: "`Integer Expression`"

```
(:operators <operator-name>*)
(:has-property <property-name>)
(:predicate <predicate>)
(:member-variable <variable-name>)
(:or <test-spec>* )
(:and <test-spec>* )
(:not <test-spec>)
```

Table 4.9: Operators for Operand Class Predicate Definition

Pan's analyzers and other infrastructure mechanisms incrementally maintain lists of nodes in several important categories. When an operand class can be specified completely in terms of the union of one or more such lists, this information may be supplied as an optional "extensional definition." Sometimes both a predicate and extensional definition are possible. This is especially helpful when the predicate may be evaluated at a single tree node more efficiently than checking for the node's membership in the membership lists.

Two implementation techniques address efficient enumeration of operand class members. First, the membership count and list of members are cached to eliminate redundant computation. This is especially helpful when only a predicate has been specified and members may be enumerated only by a complete tree walk. The second is to exploit a class's extensional specification directly when available.

4.2.7 Structural Navigation

The kernel supports two kinds of location: textual location, a single character in the text stream, and structural location, a node of the internal tree representation. The **structure cursor**, like its textual counterpart, moves via navigation primitives that take it forward or backward. Unlike text cursors (one of which is owned by every text-based window), all views on an edit object share a single structural cursor. A simple notification protocol permits any view to set the cursor and ensures that all views are notified when it changes.

Tree Traversal

Structural navigation is implemented as preorder traversal, both forward and backward, parameterized by an operand class (and in particular, by the predicate definition for the class). Thus, the primitive structural move is of the form "move to the next (previous) node for which a predicate is true." Other simple movements, for example "first," "last," "nth next," "nth previous" and "search," use the same underlying mechanism.

Many language-based editing systems use preorder traversal in some form, but often in combination with other traversals. For example, the Synthesizer Generator supports four separate commands for moving forward by one, with overlapping but subtly different behaviors. Because navigation in *Pan* is parameterized in concrete user terms (e.g. "Statement") and not by grammatical peculiarities (e.g. "Optional Placeholder"), a single method has sufficed so far. Preorder traversal has the additional advantage of reversibility, unlike ALOE's traversal, for example, whose asymmetry Medina-Mora mentions as a problem [83].

Location Mapping

A small set of conventions guide conversion between textual and structural locations. A structural location maps to the first character of the node's textual yield. The reverse mapping from text to structure is more complex, in part because, like primitive structure motion, it is parameterized by an operand class, and in part because it is parameterized by a directional bias (either "forward" or "backward"). Text to structure mapping takes places in stages:

1. Map from a character to an enclosing lexeme, if any, otherwise to the next (previous) lexeme (depending on bias), if any, otherwise to the previous (next) lexeme, otherwise signal an error (empty lexical stream).

2. If the lexeme is not a leaf node (a comment, for example) move forward (backward) in the lexical stream to a leaf node, if any, otherwise move backward (forward) to a leaf node, if any, otherwise signal an error (null tree).

3. If the leaf node is not a member, move to the nearest enclosing node that is a member, if any, otherwise traverse the tree forward (backward) to a member node, if any, otherwise traverse the tree backward (forward) to a member node, otherwise signal an error (empty class).

A similar but simpler set of conventions guide conversions between arbitrary structural locations and a desired location parameterized by some operand class (for example, to support a move to the "next statement" when the structure cursor is on a declaration).

4.2.8 View Frameworks

The low-level implementation of every *Pan* view is provided by a CLOS view class and associated methods. Figure 4.5 on page 47 depicts the current inheritance hierarchy for view classes.[12] This section describes selected elements of that hierarchy in more detail.

Standard View Classes

The class **`$primary-text-view$`** supports conventional text editing. Its language-specific descendents (offspring of class **`$lb-text-view$`**) implement the kind of language-based editing that is the primary focus of this research. Most of this dissertation describes primary text views.

In contrast, view classes **`$dired-view$`**, **`$view-list-view$`**, **`$man-page-view$`**, and **`$man-apropos-view$`** implement useful special-purpose views: a directory editor, the list of views active during a session, and a UNIX "man" page browser. These prototype implementations exploit none of *Pan*'s language-based machinery, but they emulate some of it and are designed to suggest the kinds of broad applications of the editing metaphor suggested by Fraser [36,37,38] and explored by general frameworks such as Notkin's Agave [93] and Scofield's Voodoo [112]. *Pan*'s successor, the Ensemble project [48], is exploring general applications of *Pan*'s language-based technology.

The remainder of this section describes view classes that act as implementation frameworks for the construction of useful alternate views (as opposed to the primary, editable view) onto language-based edit objects in *Pan*.

Batch Text Views

The simplest alternate view framework is implemented by class **`$batch-text-view$`**; it supports the non-editable display of a single text segment, generated anew after each incremental analysis. Any subclass that implements CLOS methods **`view-compute-name`** and **`batch-text-view-fill`** (generate the textual contents of the view) is a fully functional *Pan* view implementation. Shared behavior supported by this view class includes initialization,

[12]This hierarchy is for implementation only; to users every view is an instance of a view style; see Section 4.3.3

marking the view "dirty" when the edit object is inconsistent with its text stream,[13] and invoking method **`batch-text-view-fill`** when needed.

Like all other text-based views in *Pan*, both primary and alternate, standard text editing services are always available. Batch text views differ from primary views in this regard only by the fact that the textual contents are permanently protected against user modification.

Simple batch text views have been most useful for the continuous presentation of debugging information, for example the textual portrayal of *Colander*'s local database (**`$database-view$`**) and of the text stream's internal data structures. Data more closely related to the internal tree representation is better presented as a tree projection.

Tree Projections

View class **`$tree-list-view$`** supports the non-editable display of a sequence of text elements, each generated from a tree node member of some operand class. This amounts to a projection of some operand class onto a sequential list, modulated by a node printing method and possibly by a sort order. Class instances are fully functional view implementations, with their behavior determined by assignment to the slots listed in Table 4.10.

`operand class`	Specifies which tree nodes to project.
`print function`	Generates text entry from each tree node.
`node sort function`	Optional sorting function on member tree nodes.
`string sort function`	Optional sorting function on generated text.

Table 4.10: Parameters for Tree Projection Views

View behavior differs from batch text views by the addition of structural navigation, using the structure cursor shared with all other views on the edit object. Structural navigation proceeds forward and backward over a list, where each member is the text corresponding to a single node in the internal tree representation. When the structure cursor lands on a node, perhaps by navigation in some other view, not present in the list, then no structure cursor is visible in a projection view.

This view class framework supports a number of views useful to users as well as some useful for debugging (for example, one debugging view class projects the entire tree, using an operand class defined by the predicate `t`).

Graphical Trees

A rather different framework for alternative viewing is implemented by the class named **`$program-graph-view$`**, which uses the graphical rendering mechanisms mentioned in Section 4.2.2. Standard behavior for this class is to depict graphically *Pan*'s internal tree representation, labeling each node with text generated by a print function that parameterizes the view. These views are useful primarily for debugging *Ladle* and *Colander* descriptions,

[13]As with primary views, batch text views are shown to be dirty by a shift to alternate color maps for text rendering, see Section 4.3.9.

using print functions that reveal various aspects of the representation's internal state (parse changes, for example, and database links).

Prototype Program Presenter

An experimental text-based view framework provided powerful presentation options in a configurable way [16]. Exploiting *Colander* for access to information enabling context-sensitive formatting as well as for general purpose node attribution and propagation, this framework anticipated development now taking place within *Pan*'s successor, the Ensemble project.[14]

4.2.9 Configuration and Extension

Central to the kernel is a rich set of facilities for configuration, customization, prototyping, and extension. These facilities, summarized here, directly address *Pan*'s requirement for "extensibility and customizability" and indirectly address most other requirements. More details appear in the Version 4.0 user manual [31] and in *Pan* itself via its extensive self-documentation system.

Extension Language

Pan's extension language is embedded in COMMON LISP, deliberately blurring boundaries among system configuration, declarative customization, extension by programming, and the standard implementation of many high-level services: all these are accomplished with COMMON LISP code distinguished by heavy use of extension-level primitive functions. This open approach has proven effective for a research-oriented implementation cycle in which new services are prototyped largely at the extension level and subsequently migrate downward for improved efficiency and integration.

For example, the definitional forms (implemented as COMMON LISP macros) listed in Table 4.11 create system entities, both user-visible and internal, and register these entities with *Pan*'s internal documentation and help system. Some of these entities are described in more detail below.

All definitional forms, as well as the help system are dynamic; entities can be predefined, or they can be added and redefined at any time. Table 4.12 lists the number of instances of various entity types that exist during a session in which eleven language-based view styles have been loaded.

Pan's standard configuration is specified by a file of COMMON LISP code that is distinguished only by relative heavy use of the binding forms listed in Table 4.13, the same forms that a user might use for customization in a personal startup file.

Scoped Variables and Context

Much of *Pan*'s configurability depends on **editor variables**; these generalize COMMON LISP's global variables (normally defined using **defvar**) with bindings that depend upon **editor context**. All activity in *Pan* takes place in the context of a view (which owns a

[14]This framework does not appear in the current implementation hierarchy because its implementation was not integrated well enough to merit release with the *Pan* system.

`Define-Char-Set`	`Define-Macro`
`Define-Command`	`Define-Operand-Class`
`Define-Command-Parameter-Initializer`	`Define-Operand-Command`
`Define-Constant`	`Define-Option-Variable`
`Define-Documentation-Type`	`Define-Variable`
`Define-Function`	`Define-Variable-Cache`
`Define-Highlighter`	`Define-View-Class`
`Define-Hook`	`Define-View-Style`
`Define-Hook-Function`	`Defvariable`
`Define-Language-View-Style`	

Table 4.11: Definitional Forms in the Extension Language

Character sets:	6
Commands:	4424
Command parameter initializers:	3
Constants:	1
Documentation types:	9
Flags:	101
Functions:	153
Highlighters:	10
Hooks:	10
Hook functions:	11
Macros:	53
Menus:	129
Operand classes:	72
Operand commands:	16
Options:	98
Variable notifiers:	90
Variables (other than options):	51
View classes:	24
View styles (language-based):	11
View styles (other):	25

Table 4.12: Typical Population of Defined Entities

`Add-Hook-Function`	`File-Type-Use-View-Style`
`Add-Language-Auto-Require`	`Key-Binding`
`Auto-Require-Command`	`Modify-Collection-Variable`
`Auto-Require-Edit-Object-Class`	`Rebind-Menu`
`Autoload-File`	`Remove-Hook-Function`
`Bind-Key`	`Set-Char-Match`
`Bind-Menu`	`Set-Char-Set`
`Bind-Oplevel-Command`	`Unbind-Variable`
`Bind-Oplevel-Command-List`	`Variable`
`Bind-Oplevel-Menu`	`Window-Modify-Flag-Collection`
`Copy-Menu`	`With-Variable-Binding`
`Create-Menu`	`With-View-Scope`
`Def-File-Type-Edit-Object-Class`	`With-View-Style-Scope`
`Define-Structural-Oplevel-Commands`	`(setf (variable <var> <scope>) <value>)`
`Empty-Hook`	

Table 4.13: Binding Forms in the Extension Language

table of variable bindings), and every view has an associated view style (which also owns a table of variable bindings). Thus, context (normally defined by the locus of the user event being handled) always distinguishes three tables of bindings in the scoping hierarchy listed in Table 4.14 (the global table is shared by all view styles). Variables may be bound,

Scope	*Characteristic bindings*
`global`	Ubiquitous system behavior and defaults.
`view-style`	Preconfigured, specialized editing contexts.
`view`	Instance-based overrides and prototypes.

Table 4.14: Context-sensitive Scope Hierarchy for Editor Variables

unbound, or explicitly shadowed at any or all scope levels; inner (lower) bindings implicitly shadow outer bindings.

Variables may be set-valued, in which case individual set members may be selectively added, deleted, and shadowed in particular scopes. New variables may be defined at any time. Variables may have restrictions, for example predicates for type checking prospective values and binding restrictions to certain scopes. Editor variables may be made *active* by dynamically adding one or more **notifier** functions; these are called any time a binding occurs.

Any variable may have a **variable cache**, an automatically created "shadow" variable whose bindings are guaranteed to parallel those of the primary variable, but with a value computed by applying a specified translation function to the primary variable's value. Variable caches permit certain kinds of configuration data to be maintained in both external (user-accessible) and internal (e.g. preprocessed or compiled) forms. Translation is lazy, making variable caches especially useful for configuration data (color names as strings, for example) whose translation (to an integer color handle returned by registration with a window server) cannot be performed until run time.

All editor bindings (Table 4.13) are implemented with editor variables and are similarly scoped. Many fundamental mechanisms are parameterized dynamically by scoped **option variables**, for example fonts, colors, and window configuration.

Pan's editor variable mechanism is more general than its GNU EMACS counterpart [121]. For example, EMACS variables have only two scope levels; any mode-specific configuration must be performed by establishment of mode-specific buffer-local variables upon creation of each buffer instance. *Pan*'s **`view-style`** scope captures and reifies the notion of a mode, making it a shared, first-class entity. At the same time, the instance-specific **`view`** scope permits local overrides when needed.

Function, Command, and Macro Definition

Primitive operations in *Pan*'s extension language are implemented as COMMON LISP functions and macros, but with additional properties that ensure robustness. First, these primitives are automatically integrated with *Pan*'s undo system. Those functions (called **commands**) that are bindable to keystroke sequences and menu entries are also automatically integrated into the run-time command dispatch mechanism. Declaratively specified **parameter initializers** define the arguments needed by commands. Initializers direct the command dispatcher to collect values dynamically (by user selection, by configurable prompters, or by default) with automatic type checking and error recovery. Initializer functionality is integrated so that commands may also be called as extension-level functions by client code using no special syntax. Commands may have restrictions, for example limiting their invocation to the contexts of particular view classes or object classes.

Generic Exception Handling

A uniform framework for exception handling permits authors of simple extensions to ignore the issue and still produce robust code. Extension-level primitives guard their own internal state and signal exceptions (with an associated message) using one of the forms listed in Table 4.15. Response to these forms is sensitive to *Pan*'s exception-handling context, another

Form	*Abstract functionality*
`Announce`	Supplies information.
`Editor-Warn`	Draws attention, supplies information.
`Editor-Error`	Interrupts operation, supplies information.

Table 4.15: Forms for Signaling Exceptions in the Extension Language

of *Pan*'s several, generally orthogonal notions of internal context. In particular, exception-handling context is independent of the "editor context" used for variable scoping. For example loading a faulty language-based view style might produce a call to `Editor-Error`, but view style loading can take place in several exception-handling contexts:

- View styles may be preloaded while a new *Pan* is being built; in this case the error handler terminates the build and logs the message.
- View styles may be loaded automatically at run time; the run-time error handler displays the message, beeps, and resets the command dispatcher, effectively terminating the command that triggered the load.
- View styles may be loaded directly from within a COMMON LISP debugging loop; here the error handler calls the debugger recursively, preserving the call stack for inspection.

Exception-handling context is implemented by dynamically-bound exception handlers [130].

Integration with Help System

Implementation of every configuration mechanism in *Pan* has been accompanied by corresponding additions to the internal help and self-documentation system. These additions typically come in pairs: one command to display the configuration relative to a particular view (reflecting inherited and shadowed bindings from parent scopes), and another to list all entities of a particular kind together with internal documentation. All help system commands produce multiple-font displays that are easy to read.

This concludes the description of *Pan*'s kernel layer. The services in this layer, invisible to users, help construct the basic, user-visible services described in the following section.

4.3 The Elements of User Interaction

The basic services layer in *Pan*'s design framework (Figure 3.7 on page 30) implements the functional behavior of the system as seen by users. Following *Pan*'s requirement for "coherent user interaction with system and programs", this layer is designed to present users with a system model that is:

- *simple*, containing as few distinct new concepts[15] as possible;
- *natural*, exploiting concepts with which users are already familiar;
- *flexible*, enabling each basic service to be applied to a variety of application areas; and
- *language-independent*, providing equivalent behavior in the context of every language used.

This section describes every basic service that contributes to the system model presented to *Pan*'s users, with frequent reference to the design metaphors, presented in Section 3.5, that guide their collective design. Each basic service is implemented upon one or more of the kernel mechanisms described in Section 4.2, and in cases where a basic service is closely related to an underlying kernel service, the discussion here points out crucial differences

[15]The notion of *new concept* is relative, of course. The intended sense here is "new relative to experienced software practitioners" of the kind discussed in Chapter 2. Those practitioners are presumably familiar with at least one powerful text editing system.

between the two. Every basic service appears to the user only in the context of a particular configuration; the discussion here mentions important degrees of freedom for configuration in each case, and Section 4.4 discusses the general issue of configuration more thoroughly.

How basic services are *used* is of paramount importance, but the discussion here centers on their definition and behavior, mentioning only suggestive examples of their use. Chapter 5 on the other hand is dedicated exclusively to the topic, showing how *Pan*'s basic services can serve as building blocks for a wide variety of applications, and how they, when appropriately configured, resolve a number of problems with user interaction.

4.3.1 Multi-Window Text Editing

Pan's text editing interface is a bit-mapped, multiple-font, mouse-based system with multiple windows, in the spirit of Bravo [70] and its many successors. A *Pan* user may open any number of **windows** onto each document's virtual two-dimensional text display. All windows on a document share a single, visible **selection** that appears as underscored text in any window in which it happens to be visible. Each window has its own scroll position and text **cursor**, both of which persist when the window is made invisible. Some text-oriented commands operate on the selection, and others, including ordinary character insertion, operate at the text cursor; all are undoable.

The editor is a hybrid based on two familiar models: the Macintosh [108] and EMACS [122]. Like both of those and many other editors, *Pan* treats text as both a stream and a two-dimensional page. Like Macintosh editors *Pan* has an insertion point that is distinct from the selection,[16] has a global **clipboard**, and supports the menu- and key-driven commands `Cut`, `Copy`, and `Paste`. Like EMACS, *Pan* offers a rich set of text-oriented editing commands and EMACS-compatible keybindings [31].

4.3.2 Simplified View Model

Although views are implemented by the internal object/view model, described in Section 4.2.5, *Pan* presents the user with a much simpler model that is no more complex than an ordinary editor much of the time.[17] In this simplified model there exists mainly windows and files (along with other named things, for example, directories, and man pages) that a user can "visit," where "visit" means "open a window onto". Of the four concepts in the internal model (edit object, buffer, view, and window), the notions of edit object and buffer are concealed entirely, and the notion of view is implicit much of the time.

Hiding Edit Objects in the Model

From the user's perspective, "edit object" merely names the category of things that can be visited in *Pan*, an unnecessary bit of implementation terminology. All communication between *Pan* and user about edit objects is phrased in terms of specific instances, for example "The `File fact.mod`" or "The `Directory src/`."

[16]This conflicts with the EMACS model where the insertion point *defines* one boundary of the selection.

[17]This approach derives not from any condescending estimate of a typical user's ability to understand the full model, but from the rather opposite point of view: the typical user has more important and complex matters to think about than useless (to the user) implementation concepts that the designer was unable to abstract away.

Hiding Buffers in the Model

As depicted in Figure 4.3 on page 45, a *Pan* buffer embodies two relationships important to users; both are revealed by naming conventions, with no mention of buffers. The relationship between the internal edit object and (possibly) a persistent storage location, is captured by the name of the buffer, which is also the name of the default primary view that the user sees. That name follows familiar conventions. For example a buffer on the file stored in `/usr/local/src/manual.tex` might be named "`manual.tex`" unless there is another buffer of the same name, in which case "`manual.tex<2>`" is used. As with GNU EMACS and many other editors, users are free, in fact encouraged, to think of the short name as a convenient handle (or surrogate) for the file itself, with no consideration of the machinery involved. The second relationship, between the internal edit object and one or more views, is discussed next.

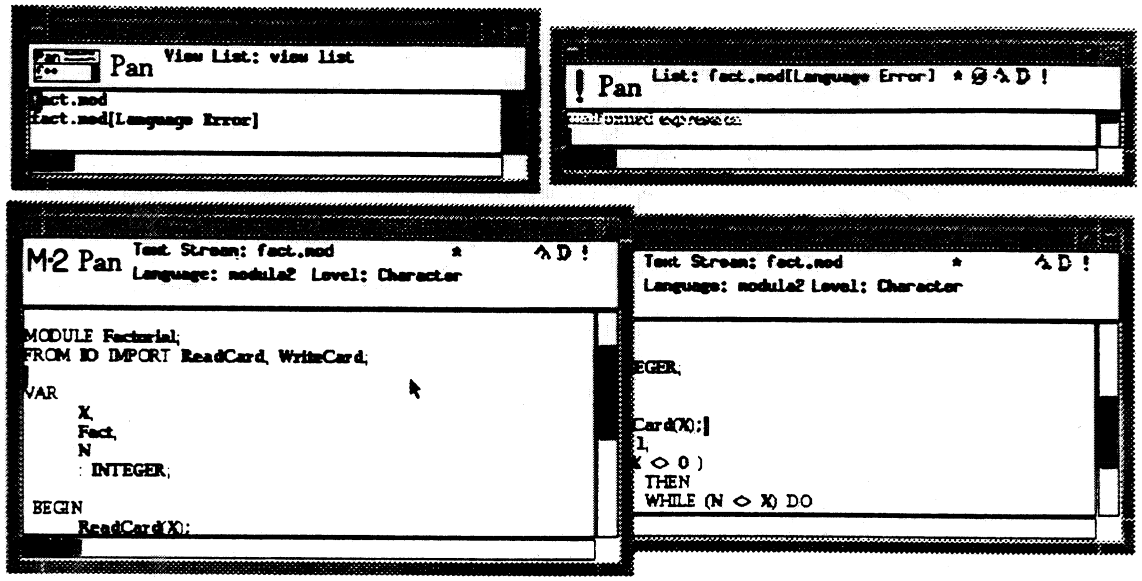
Downplaying Views in the Model

In the ordinary case a user invokes the *Pan* `Visit` command, supplies the name of a file, and a window appears showing part of the file's contents. The user may open more windows as needed, for example as depicted in Figure 3.2 on page 23. Each window is labeled in terms of the specific thing being visited. For example in the window labeled "`Text Stream: manual.tex`," the string "`manual.tex`" is the view (and buffer) name and "`Text Stream`" is the kind of view. The name "`manual.tex`" also appears in a special window that lists the short names of what is currently being visited. That special window is titled "`View List`" and is the only explicit mention of views. The user opens windows on existing views by simply clicking on a name in the view list and invoking the command `Visit`. In the simple case the user is encouraged to think of the short view name as a surrogate for the file itself; in effect, the edit object, buffer, and primary view all have the same name.

Potential for confusion arises when the user requests alternate views in addition to the primary view, since the relationship between edit objects and views is now one-to-many and this relationship must be clear to users. A new mechanism could be added, for example windows subdivision or a more complex view list, but those carry both implementation costs and cognitive overhead for users. A simple extension to *Pan*'s view naming convention, along with careful choice of view names, suffices, as shown in Figure 4.7.[18] Just as a numeric suffix of the form "`<2>`" disambiguates edit object names when necessary, a suffix naming the kind of view, "`[Language Error]`" in the figure, disambiguates view names when necessary, that is, when a buffer has more than one view. The view suffix is null for primary views, ensuring that the multiple view mechanism is invisible in simple cases.

Uniqueness of view names isn't enough, however; users must be able to answer two crucial questions at a glance. First, "what (edit object) is this window associated with?" This is evident from the first part of the view name, which names the thing being visited (buffer and edit object). Second, "what views do I have (associated with this edit object)?" This is evident in the "`View List`" view; all view names are sorted alphabetically, effectively clustering related views (with alternate views following the primary view).

[18]The diagram appearing in Figure 4.3 on page 45 shows this scenario as reflected in implementation concepts.


Pan
View List: view list
! Pan
List: fact.mod[Language Error] * ⚠ ↰ D !
M-2 Pan
Text Stream: fact.mod * ↰ D !
Language: modula2 Level: Character
Text Stream: fact.mod * ↰ D !
Language: modula2 Level: Character

```
fact.mod
fact.mod[Language Error]
```

```
malformed expression
```

```
MODULE Factorial;
FROM IO IMPORT ReadCard, WriteCard;

VAR
      X,
      Fact,
      N
      : INTEGER;

 BEGIN
      ReadCard(X);
```

```
EGER;


Card(X);
1,
X <> 0 )
 THEN
 WHILE (N <> X) DO
```

Figure 4.7: Using *Pan*: Multiple Views

4.3.3 View Styles

In contrast to *Pan*'s object/view model (Section 4.2.5) and implementation of view classes (Section 4.2.8), the **view style** is a first-class, user-visible concept. A view style is a *mode* in the general sense: it modulates how the system responds to user actions. A view style in *Pan* is a context for user interaction that offers a particular configuration of services designed to assist a particular class of users with a particular kind of task involving a particular kind of document.

Modes are the subject of some controversy and confusion [62,128], but they are ubiquitous in real life [61] and necessary for powerful systems such as *Pan*. As a practical matter, modes invite a large class of user errors that Norman calls "mode errors": user action appropriate to a system state (mode) other than the one the system is in at the time [91]. The best design strategy for ameliorating these problems, other than to avoid needless modes and gratuitous variation, is to provide *mode feedback* so that relevant system states are continually visible.

Since *Pan* permits creation and loading of any number of different view styles, mode feedback for view styles is configured by the view style specification. A specified **view style logo** appears at the top left of each window's panel; this is a visual announcement and reminder of the view style in effect for the window. *Pan*'s standard configuration assigns a default view style to each kind of edit object that the user might visit, so the view style logo serves simultaneously to identify the view style and the kind of thing visible in the window, for example plain text file, directory, or Modula-2 program. As a further contextual clue, the **language name**, appearing in the panel, displays which underlying language, if any, is being used by the view style. In cases where only one view style exists per language, this is redundant (though useful), but in general there may be more than one view style based on an underlying language.

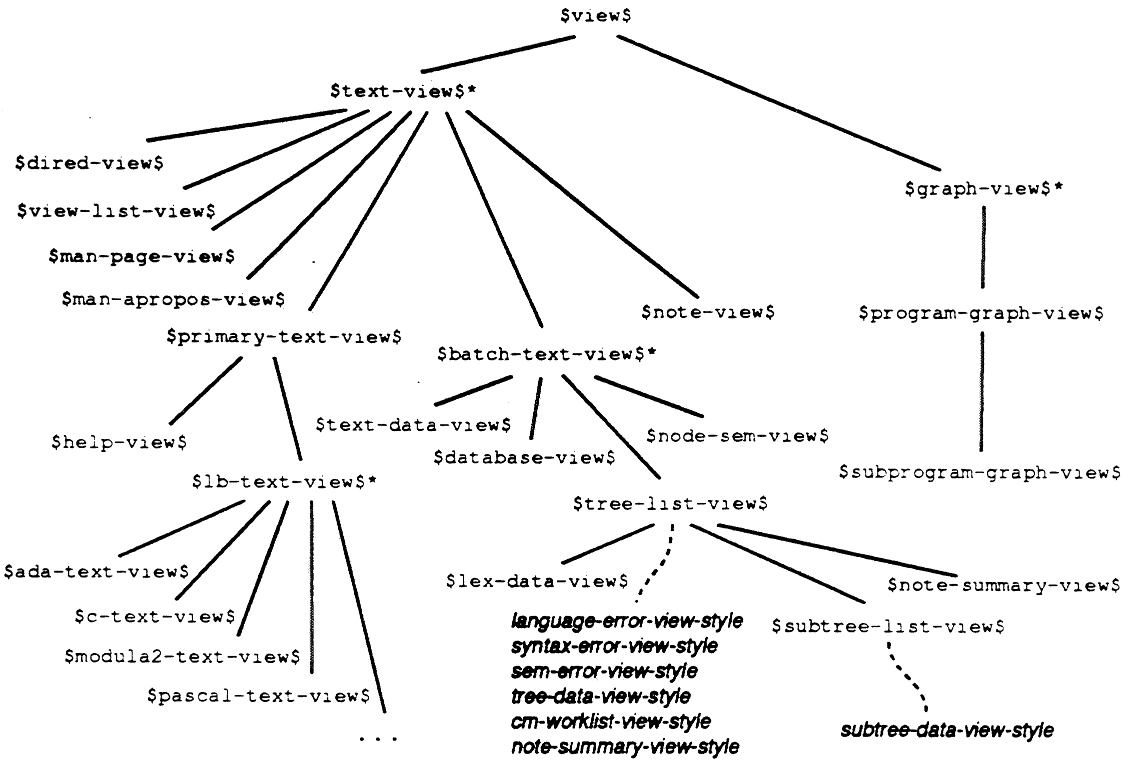


Figure 4.8: View Classes and View Styles

Pan view styles are implemented primarily by configuration, using the scoping mechanisms described in Section 4.2.9 to create and configure instances of various basic services. A user familiar with the notion of view style configuration can obtain detailed information at any time about a particular view style in effect (taking into account local, view-specific overrides) by invoking *Pan*'s help system (Section 4.3.12).

A view style may optionally specify an underlying language, whose *Ladle* and *Colander* descriptions will be loaded and used automatically in the context of the view style. Section 4.4 discusses techniques and guidelines for creating effective language-based view styles in *Pan*.

An essential part of each view style's configuration is selection of a view class to implement its general behavior. Figure 4.8 shows a subset of the current class implementation hierarchy for views (as in Figure 4.5 on page 47) annotated to show how view classes implement view styles. Not shown are so-called "mixin" classes that exploit multiple inheritance to add implementation features. View classes without annotation in the figure currently implement exactly one view style, usually similarly named. For example, the view style **view-list-view-style** is the only view style that uses class **$view-list-view$**. Some classes on the other hand, for example view class **$lb-text-view$**, do not implement any view styles directly, but support subclasses who share behavior; these classes are annotated with an asterisk in Figure 4.8. Finally, some view classes are shared by several view styles, since the difference in their behavior can be specified completely by configuration; these classes are annotated with lists of client view styles in the figure.

4.3.4 Panel Flags

Just as the view style logo at the top left of each panel provides important contextual information, an array of **flags**, small glyphs at the top right of each panel, supply contextual information of a more dynamic sort. Modeled loosely on indicator (or annunciator) lamps on physical control panels, panel flags may appear and disappear, change shape, change color, or all three in response to a change of state concerning the document. A flag has the properties listed in Table 4.16.

`variable name`	Name of scoped variable whose value determines state.
`on bitmap`	Image when on.
`on foreground color`	Image foreground color when on.
`on background color`	Image background color when on.
`off bitmap`	Image when off.
`off foreground color`	Image foreground color when off.
`off background color`	Image background color when off.

Table 4.16: Panel Flag Properties

In Figure 3.2 on page 23, for example, the flag whose image is "*" appears, indicating that the document has been changed since last saved; otherwise this flag is configured to be invisible. The appearance of the flag follows the value of a variable named `Object-Modified?`. The flag "↩" also appears, indicating that automatic text-filling (line-wrapping) is in effect. Other flags might reveal whether certain kinds of inconsistency have been detected, or any other property that can be expressed in terms of an editor variable (Section 4.2.9). A common flag configuration has an image appearing only when members of a particular operand class are present ("`Placeholder`", for example, as described in Section 5.5, or "`Language Error`" as described in Section 5.4).

4.3.5 Highlighters

Just as a panel flag can be configured to reveal whether any members of an operand class are present, a **highlighter** can be configured to make its members visible. A highlighter embodies a binding between an operand class and a text effect applied to the textual display of members; it has the properties listed in Table 4.17. The range of possible effects is limited by the internal text representation and text rendering engines described in Section 4.2.1: three (non-black) ink colors, and two independent background colors for shading.

Highlighters represent one of many basic services configured and implemented by the kernel's operand class mechanism. In most cases, however, the kernel concept of "operand class" is downplayed. Instead, the user views and toggles specific highlighters, for example a highlighter for "language errors" and perhaps another for "stylistic violations."

A highlighter is an example of a gracious service, one that can (when requested) operate in both exact and approximate modes. A strict highlighter only appears when a document is consistent; a non-strict highlighter persists during periods of inconsistency, but its visual effect is rendered using alternate color maps so that changes in state between exact and approximate are visible.

`name`	A symbol e.g. `pascal-int-expr-highlighter`.
`title`	A textual name for users, e.g. "Integer expressions".
`documentation`	A string.
`apropos`	A list of subject words, e.g. "integer," "expression."
`opclass`	A symbol opclass name e.g. `pascal-int-expr-opclass`.
`effect`	One of `:fg1 :fg2 :fg3 :bg1 :bg2`.
`strict?`	Either `t` or `nil`.
`toggle-command`	A symbol command name, e.g. `Toggle-Int-Expr-Highlighter`.

Table 4.17: Highlighter Properties

```
(:highlighters (lang:lang-err-highlighter :on  "^C ^C !")
               (expr-highlighter          :off "^C ^C e")
               (lang:query-highlighter    :on  "^C ^C q")
               (lang:cursor-highlighter   :on))
```

Figure 4.9: Example Highlighter Specification

Definition of a highlighter also results, by default, in the creation of a *Pan* editor command that toggles the state of the highlighter between active and inactive in the current view. This command may be bound to menus and keystroke sequences. When one of these commands is invoked, the display is updated immediately and an announcement appears of the form "`Highlighter <title> is now ON [or OFF]`." Keybindings for these toggle-commands are bundled together with the specification for the set of highlighters that are available for the view style and the initial state of each. For example, in a sample view style named "Simple," the user may select from among four highlighters, three of which are initially enabled in every new view, as configured by the specification fragment appearing in Figure 4.9. Nothing prevents a view style designer from highlighting two classes the same way. Nothing prevents a view style designer from highlighting one class in two different ways simultaneously.

4.3.6 Operand Level

Each *Pan* window displays a current **operand level**, which the user selects from a panel menu of operand classes specified for the view style. The operand level is a very weak input mode that effectively modulates the operation of a few generic commands; Table 4.18 lists these level-sensitive commands. The operand level affects no other commands, and a user may choose (via menu- and keybindings) not to use the level-sensitive versions at all. In particular, the operand level neither inhibits nor modulates text-oriented editing. When the current level is "`Statement`", for example, the user may invoke `Oplevel-Cursor-To-Mouse` (this is the "smart" version of the usual text cursor-setting command). This sets the cursor to the "nearest" structural component that meets the definition of operand class "`Statement`", mapping locations by coercion, if necessary, as described in Section 4.2.7.

`Oplevel-Select`	`Oplevel-Cursor-In`
`Oplevel-Cursor-To-Mouse`	`Oplevel-Cursor-Out`
`Oplevel-Mouse-Extend`	`Oplevel-Cursor-Search-Forward`
`Oplevel-Query-All`	`Oplevel-Cursor-Search-Backward`
`Oplevel-Cursor-Forward`	`Oplevel-Delete`
`Oplevel-Cursor-Backward`	`Oplevel-Replace-All`
`Oplevel-Cursor-To-First`	`Oplevel-Query-Replace-All`
`Oplevel-Cursor-To-Last`	`Oplevel-Menu`

Table 4.18: Operand Commands

An operand level is implemented a binding that maps user actions into command invocations deemed appropriate (by view style configuration) to a particular operand class. An operand level has the properties listed in Table 4.19. Definition of a level also results,

`opclass`	A symbol opclass name e.g. **`ada-stmt-opclass`**.
`bindings`	A map: generic command `->` class specific command.
	A map: generic menu `->` class specific menu.
`before-daemon`	A function, run before each command dispatch.
`after-daemon`	A function, run after each command dispatch.
`set-command`	A symbol command name, e.g. `Set-Oplevel-To-Ada-Stmt`.

Table 4.19: Operand Level Properties

by default, in the creation of a *Pan* editor command that makes the level "current" in the active window. This command may be bound to menus and keystroke sequences and is an alternative to level selection with the window's panel menu. When one of these commands is invoked, the level showing on the panel changes, and an announcement is made of the form "**`Changing level to <level>.`**" For example, in view style "simple" the user may select from a list defined by the fragment of its view style specification appearing in Figure 4.10, using either the specified keybindings or by selection from the panel menu.

Separate binding specifications establish the maps between keystroke sequences and commands. For example the level-sensitive command **`Oplevel-Cursor-Forward`** typically maps to command **`Next-Character`** at level "**`Character`**" but to **`Next-Oplevel-Node`** at all language-based classes such as "**`Statement`**" and "**`Expression`**". **`Next-Oplevel-Node`** is the standard predicate-based tree navigation primitive described in Section 4.2.7; the oplevel dispatcher parameterizes it with the predicate that defines the operand class of the current level. The level-sensitive command **`Oplevel-Menu`** displays a menu of currently available level-sensitive commands, and may in cases be used to produce menus even more context-dependent, for example placeholder expansion menus used in syntax-directed editing (Section 5.5). User experience with the system is surprisingly colored by keybinding patterns, as discussed in more detail in Section 5.3.

```
(:operand-levels (text:character-level    "^c c")
                 (text:word-level         "^c w")
                 (text:line-level         "^c l")
                 (lang:lexeme-opclass     "^c x")
                 (simple-expr-opclass     "^C e")
                 (simple-stmt-opclass     "^C s")
                 (simple-decl-opclass     "^C d")
                 (lang:query-opclass      "^C q")
                 (lang:lang-err-opclass   "^C !" "^C #"))
```

Figure 4.10: Operand Level Specification for "Simple"

4.3.7 The Cursor

Pan's edit cursor has two aspects: a textual location, displayed as an inverted box, and an optional location at some structural component. When positioned structurally, the textual extent (or yield) of the cursor's structural component is highlighted (see the example highlighter specifications in Figure 4.9 on page 64), and the inverted box is positioned at the beginning of the extent. The user may mix text-based and language-based cursor moves freely. Following the "Smart versus Dumb Services" design metaphor, when the cursor has a structural location it is metaphorically "smart," in the sense that language-based information was used to position it, otherwise it is "dumb." The appearance and disappearance of highlighting to the right of the inverted box indicates the cursor's current state.

Editing operations that need a cursor location use the appropriate aspect. When a structural location is needed from a text-only cursor, one is inferred using standard location mapping, as described in Section 4.2.7; this is the same coercion mechanism invoked when the user sets the cursor to a structural component by pointing with the mouse at a character.

The cursor is implemented upon two separate mechanisms from the kernel, the text cursor (Section 4.2.1) and the structure cursor (Section 4.2.7), by maintaining invariant relationships between the two. Combining the two mechanisms reduces the number of items of editor state that the user must manage and track (a form of conceptual "chunking") and permits a fluid mix of text and language-based movements.

As a further convenience in the current implementation, the cursor's structural extent is always selected textually. Thus a language-based cursor move gives the user the option to start typing at the text position or to delete textually the selected structural component under the cursor before proceeding.

The implementation and invariants described here are a snapshot of *Pan*'s evolving editing model. Experience and analysis suggest that it is still has confusing characteristics and still lacks the full flexibility needed. Some of the problems are visual. For example, the inverted box display for the text cursor is an anachronism dating from early monochrome implementations, in what the convention was already an anachronism modeled on character-based display terminals. A much better text cursor display, and one much easier to understand in terms of textual insertion, would be positioned *between* characters, as is the vertical line used by Macintosh applications or a small carat positioned at the text baseline. Those solutions would also avoid unpleasant interactions with the colormap that the current inverted box evokes.

The rest of the problem concerns the basic model of navigation and selection, and in particular the distinction between the two, supported by the system. Section 5.3 discusses the current model, its problems, and a proposal for a more suitable version.

4.3.8 The Query

User and system can use the cursor to communicate about one structural document component at a time. For example, the user can set the cursor at an identifier prior to invoking `Goto-Declaration`, and the system replies by setting the cursor structurally to the appropriate declaration. In many cases, however, one component at a time is insufficient. Had the user in the example instead invoked `Query-Def-And-Uses`, the system's reply would be a **structural collection**, implemented as a set of tree nodes in the internal representation.

Pan currently supports collection-based interaction with the "`Query`", a distinguished operand class whose membership can be set at any time by various applications, for example by `Query-Def-And-Uses`.

The class "`Query`" is commonly configured with a highlighter (see the example highlighter specifications in Figure 4.9 on page 64), so the net effect of `Query-Def-And-Uses` is to highlight the definition and all uses of a selected variable. "`Query`" is also commonly configured as an operand level so that smart navigation can be used to move from member to member. Each assignment of the query is accompanied by a name that identifies its origin, for example "`Uses of <name>`," and the level "`Query`" is typically configured with an **`after daemon`** that announces the name of the query upon each successful navigation at the level. Likewise, a panel flag could be configured that would reveal when a query is present.

As with the cursor, the query mechanism is an interim service in *Pan*'s evolving editing model. Experience and analysis suggest that it isn't flexible enough; Section 5.3 discusses the current model and explains how it can effectively be subsumed by an appropriately generalized model of selection.

4.3.9 Analysis Invocation

Pan's policy for incremental analysis is demand-driven: it takes place only when the user requests it, either *implicitly* by invoking a strict service configured to trigger analysis before proceeding, or *explicitly* by invoking `Analyze-Changes`. Nothing prevents a user from typing an entire document without once invoking analysis.

Structural navigation, for example, must be strict at present due to implementation restrictions. All smart navigation commands, for example `Oplevel-Cursor-Forward`, run the analyzers, if the document is inconsistent, before attempting language-based movement.

The distinction between the consistent and inconsistent states is explicit in *Pan*'s model, and users can know at a glance which state a particular document is in. At the same time, it is important to soften the thresholds as the state changes between the two, so the cues are non-intrusive: font shifts for new text, panel flag changes, and coloration changes for highlighters. These cues are further designed to support the "Smart versus Dumb Services" design metaphor, with the implication that the whole system is smarter when consistent, and the "Strict versus Gracious Services" design metaphor, with the implication that many basic services (the gracious ones) work almost the same in both states. These issues are discussed in more detail in Section 5.6.

4.3.10 Annotation

Pan's prototype annotation service permits the addition of textual notes that reside in the database, and which do not appear directly in a primary view. These annotations are independent of traditional language-based comments, although bridges between the two could be established.

Any structural component of a document may have assigned to it one or more named chunks of text (each chunk is implemented as an instance of the text stream representation described in Section 4.2.1). The name serves to associate related annotations. In simple cases, only a single default name might be used ("`Note`" for example) so that there would be only one kind of annotation present. More ambitious users might apply layers of annotations distinguished by name. Most usefully, applications invoked by the user might create new annotations with distinguished names; for example, a debugger interface might be requested to "load" a particular stack trace into the database by annotating the structural components involved with details about call frames, local environments, and the like.

Familiar basic services provide access to annotations:

- Ordinary structural navigation, followed by an invocation of `Visit-Note-View`, permits users to see a particular existing annotation (in an alternate view), to modify its text, or to create a new one at the cursor.

- A panel flag can be configured to appear whenever any annotations are present.

- A highlighter can be configured to reveal which components have annotations.

- An operand level "`Note`" permits smart cursor movement among annotated components. After each movement at this level, an associated after-daemon causes the first line of the notation to be announced in the panel.

- Finally, an invocation of `Visit-Note-Summary-View` (described in Section 4.3.11) produces a single read-only view containing a list of all current annotations. Selecting the text of an annotation in the summary view causes the cursor in the primary view to move to the annotated component.

4.3.11 Alternate Views

Alternate views, configured by predefined view styles, display useful information derived from a document. In most cases shared structural navigation (all views including the primary one share the same structural cursor) makes it convenient for the user to navigate document structure using several kinds of information.

Commands for visiting alternate views always check for the prior existence of an appropriate view, which is always made visible in preference to creation of a new view. As with all views in *Pan*, the user is free to open any number of windows on a view, to close all windows on a view, and to remove a view entirely.

Error List View

An **error list view** displays all language-related diagnostic messages generated by *Pan*'s analyzers. This read-only view supports shared structural navigation with the single operand level "**Language Error**" permanently in effect. Thus, the operand level mechanism is invisible in this view style, but level-sensitive operand commands for navigation listed in Table 4.18 on page 65, including pointing with the mouse, do the right thing. They select and highlight diagnostic messages; at the same time they set the cursor in the primary view to the structural location of the selected diagnostic, which in turn causes the active window on the primary view, if visible, to scroll to the new cursor location.

Cross Reference View

A **cross reference view** lists all the names (variables, for example) defined in a program, sorted alphabetically. Each name is annotated with its type and the name of the scope in which it is defined, if this makes sense in the underlying language. Setting the cursor structurally in this view moves the cursor in the primary view to the defining instance of the name selected.

Table of Contents View

A **table of contents view** lists all the top level procedure headers defined in a program, in order of appearance. Each is annotated with its argument signature, when this makes sense in the underlying language. Setting the cursor structurally in this view moves the cursor in the primary view to the definition of the procedure selected.

Note View

A **note view** displays a textual annotation associated with some structural component in the database. Unlike most alternate views, this view is editable. Setting the cursor structurally in the view highlights the entire note and sets the cursor in the primary view to the component being annotated.

Note Summary View

A **note summary view** lists the concatenation of all annotations of a particular kind. As with other list views, a smart cursor movement selects an entire annotation and moves the cursor in the primary view to the annotated component.

Prettyprinting View

An experimental family of **prettyprinting views** were developed as part of a research project that explored alternate program presentation techniques.

Graphical Tree Views

A family of commands create alternate views that graphically depict *Pan*'s internal tree representation; they differ only in the information that appears labeling each node. These are generally useful only for *Pan* developers and authors of *Ladle* and *Colander* descriptions, since they cast no useful light on program structure as perceived by a user.

Debugging Views

A number of other text-based alternate views present various kind of debugging information, usually text-based descriptions of internal data structures whose display is updated automatically with each analysis. When any relationship exists that associates displayed text with particular structural components, the relationship can support shared navigation. More general graphical display mechanisms for internal data structures are under development using this framework [98].

4.3.12 Access to System Information

Fred Hansen argued as one of his "user engineering principles" for Emily, the grandparent of all language-based editing systems [52], that:

> The user should be given access to [various parameters] and should be able to modify from the console any parameter that he can modify in any other way. With access to the system information, the user need not remember what he said and is not kept in the dark about what is going on.

Pan's help system, along with interactive versions of option-setting commands, follows Hansen's principle precisely. Table 4.20 lists commands that display various aspects of *Pan*'s configuration in any context.

4.4 View Style Design

The view style layer in *Pan*'s design framework (Figure 3.7 on page 30) configures collections of *Pan*'s basic services into editing contexts known as **view styles**. Each view style is designed to support particular tasks confronting a particular group of users using a particular underlying language. Whereas *Pan*'s implemented collection of basic services presents an intentionally simple and language-independent model of the system, each view style presents a model of document browsing and interaction that is language and situation specific. And while view style design is cast as a form of configuration, the issues involved and decisions to be made are substantial.

User-centered design is just as important for view styles as for basic services, but only users, or people working closely with them, have the local knowledge necessary for effective view style design. Experience with the SMARTSystem, a single-language commercial system that shares some goals with *Pan*, confirms heavy demand for local customization to enhance productivity [44,97].

A goal for this layer of *Pan*'s implementation framework is to make view style design and specification accessible and straightforward. As with design in general, any framework powerful enough to permit good design can only *encourage* good design. It cannot prevent bad design, but examples and guidelines can help. This section describes view style design in *Pan*: what kind of people can be expected to do it, how they express their designs, and what they should know for effective design.

`Apropos-String`	`Display-Hooks`
`Display-All-Key-Bindings`	`Display-Key-Binding`
`Display-Apropos-Index`	`Display-Key-Bindings-For-Command`
`Display-Auto-Exec`	`Display-Language-List`
`Display-Auto-Load`	`Display-Language-Specifier-Documentation`
`Display-Bug-Report-Info`	`Display-Macro-Documentation`
`Display-Char-Set-Documentation`	`Display-Notifiers`
`Display-Char-Sets`	`Display-Operand-Class-Documentation`
`Display-Command-Documentation`	`Display-Operand-Command-Bindings`
`Display-Commands`	`Display-Operand-Command-Documentation`
`Display-Configuration`	`Display-Operand-Level-Bindings`
`Display-Constant-Documentation`	`Display-Operand-Level-Menus`
`Display-Default-Menu`	`Display-Option-Documentation`
`Display-File-Ids`	`Display-Option-Variables`
`Display-Flags`	`Display-Trace-Status`
`Display-Function-Documentation`	`Display-Variable-Documentation`
`Display-Highlighter-Documentation`	`Display-Version`
`Display-Highlighters`	`Display-View-Class-Key-Bindings`
`Display-Hook-Documentation`	`Display-Viewed-Objects`

Table 4.20: Help System Access to Configuration Data

4.4.1 The Role of the View Style Designer

View style design in *Pan* must be carried out in the context of intended use, where information about particular groups of users, their tasks, and the way they use their languages is available. View style designs developed along with the system itself can only serve as examples.

View style design cannot be automated in the sense of Mackinlay's APT system [81]. View style design in *Pan* has more degrees of freedom, and is more sensitive to context than APT. A presentation design generator might be specialized for this task via deeper domain-based reasoning (e.g. the model-driven presentation design system [5]), but even if the power of these systems were sufficient, local designers would still have to supply the system with enough information about the local context to make the results appropriate. There remains an inescapable need for site-specific expertise.

Experience shows that powerful systems used at large sites tend to attract individuals who are representative users, but who also become adept at configuration and extension. People in this role have been called variously "local developers" [41], "translators" [80], and "tinkerers" [78], in recognition of their potential contribution to the local community. This role is becoming more formal and well-supported, and is seen to be essential to increased productivity, since it involves both considerable technical sophistication and working knowledge of a user population and their tasks [41]. These are precisely the people who would modify and adapt sample *Pan* view styles to meet specific needs (other than low-level language-description work).

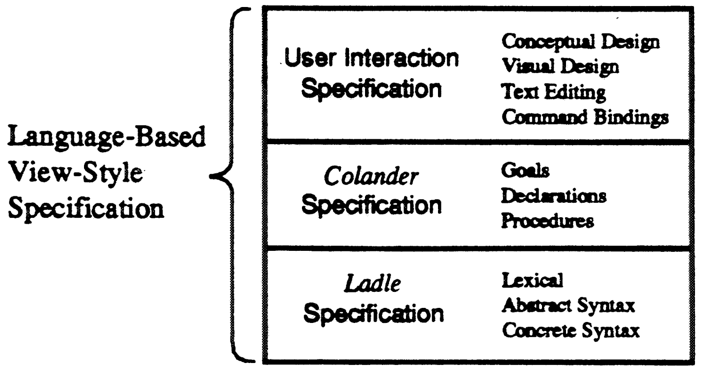


Figure 4.11: Specification Layers

4.4.2 Specification Layers

A complete view style specification in *Pan* contains three parts, whose relationship is suggested by Figure 4.11. Each layer depends on at least some of the specifications in lower layers; a preprocessor for each produces data used by parts of the run time system, as depicted in Figure 4.1 on page 36.[19] From the perspective of a view style designer, there are important degrees of freedom in each layer.

The *Ladle* Specification

The *Ladle* specification, introduced in Section 4.1.2 and described more thoroughly by Butcher [22], contains a lexical specification, an abstract syntax specification, and additional concrete syntax specifications to enable parsing. Most important to the view style designer is the abstract syntax, which together with directives on grammar rules, permits considerable control over *Pan*'s run-time internal tree representation.

Experience shows that tree representation choices are best made in conjunction with the *Colander* specification, since most of *Colander*'s analysis is expressed in terms of goals at tree nodes and node properties. Choices include abstraction (removal) of nodes representing chain rules, use of explicit sequence nodes in place of recursion, and the operator names at tree nodes.

Structural choices can be made without concern for usability precisely because of the user interaction layer, which frees the *Ladle* description from the kinds of user-motivated demands on AST design suggested for Mentor [29,63]. The view style designer need only be aware of the general structure and operator names in order to specify appropriate operand classes.

[19]Sharing of lower layers by more than one upper layer is desirable and theoretically possible, but current implementation restrictions permit this only by copying.

The *Colander* Specification

The *Colander* specification, introduced in Section 4.1.3 and described more thoroughly in Ballance's dissertation [9], consists primarily of goals to be satisfied at each node in the internal tree representation. Other specifications include computed node properties, facts asserted into the database as side effects of goal satisfaction, and procedures that implement complex database queries.

Colander's specification language offers many degrees of freedom, some relevant to the view style designer. The first decision concerns the scope of the constraints to be checked. Although common practice is to check at least all context-sensitive constraints specified as part of an underlying language's formal definition, this need not be the case, especially for specialized view styles where a subset might suffice (and might yield better run-time performance). The *Colander* specification can even be omitted entirely, in which case the user interaction specification must rely on structural information alone. Conversely, a *Colander* specification can be extended beyond language definition; experimental *Colander* specifications have checked stylistic rules, performed data flow analysis, and computed prettyprinting directives.

A view style designer must be partially aware of the *Colander* specification's database, so that the information can be used for operand class specification. Some services may require that new queries be supported in *Colander*, for example complex queries concerning names and scopes that can't be cast in terms of simple node properties. Unfortunately, additions to *Colander* specifications involve time-space tradeoffs of the sort that confront database design in general as well as interaction with other aspects of the specification. This shortcoming of the *Colander* prototype effectively places extensions depending on complex queries beyond the reach of many potential view style designers.

The User Interaction Specification

A user interaction specification configures *Pan*'s basic services, introduced in Section 4.3, to define visible behavior of a view style. The specification relies on a *Ladle* specification for structural information and on a *Colander* specification for results of *Colander*'s run-time analysis, but the model of document structure exposed to the user and the choices of available services are made in the user interaction layer.

Appendix A contains an annotated example of a user interaction specification for the language Modula-2. Appendices B and C describe the provisional language in which view style specifications may be expressed. The next section discusses issues that confront view style designers.

4.4.3 Design Issues and Guidelines

This section describes part of the design process for the user interaction layer in a *Pan* view style, beginning with general issues and progressing through document modeling, visual design, keybindings, and integration with other services. The discussion of design decisions that confront the view style designer is accompanied by specific guidelines. Chapter 5 explores the importance of these design decisions in the context of various aspects of user interaction.

Important to *Pan*'s general framework are the design metaphors, described in Section 3.5, which present a coherent system model to users. Although the collection of basic services described in Section 4.3 supports these metaphors, that support must be continued throughout view style design, revealing again the pivotal role played by designers in making the system effective in practice. The first two are meta-guidelines that apply implicitly to all that follow.

Design Guideline 1 For any target group of users, configure all view styles as much alike as possible and vary them in detail only as needed for language idiosyncrasies or the needs of services delivered. It is especially important that visual design themes, including colors, fonts, and logos, be uniform.

A typical user interacts with many view styles, one per language in the simple cases, but with more when they are specialized to support particular activities. Uniformity of interaction across multiple view styles, an important part of *Pan*'s framework, can only be realized if all view styles that each user sees are likewise uniform.

Design Guideline 2 Make all useful information available to the user redundantly through as many channels as possible, for example by highlighting, by navigation, and by projected views.

This second meta-guideline emphasizes the value of redundant encoding. In some situations this simply increases effective bandwidth, but it also allows for task variations which cause users to need information in different forms. Finally, it acknowledges variation in the abilities of users to absorb information by different means. Color-blind users, for example, might rely more heavily on projected views and navigation than on highlighting.

Conceptual Design

View style design begins with two related questions: what model of document structure to present and what services to make available that help manipulate and browse that structure. Answers depend on the intended user population, tasks, and the underlying language. For example, a novice might appreciate syntax-directed editing templates for writing new code, but an experienced user might not want to waste the menu space. Tools for re-engineering old software would be of little use for coding a small prototype. Knowledge of specialized libraries should only be included in view styles for relevant languages and particular systems in which they are used. Finally, slightly specialized view styles can help check for the kinds of problems that arise when porting compilers, language versions, and platforms.

Design Guideline 3 Define and name syntactic operand levels following local vernacular for describing document parts; use the singular.

A view specification implies a model of document structure by the membership of its operand level menu. This list of named classes forms the working vocabulary with which user and system communicate. Simple syntactic levels should reflect the way users think and talk about components. This is straightforward in many languages, where levels for "`Procedure`", "`Declaration`", "`Statement`", and "`Expression`" are usually appropriate.

Other cases require judgement, for example whether to reveal the fact that the categories called "statements" and "expressions" in C actually overlap, and whether to create separate categories in languages that formally distinguish among "definitions," "declarations," and "specifications." Logic-based languages like Prolog and *Colander*'s specification language present an entirely different set of choices. An introductory language textbook is a good starting point for building a taxonomy of document components, but refinement based on experience and personal preferences is usually necessary.

Design Guideline 4 Include standard operand levels, as needed for basic services.

Predefined operand levels, for example "`Query`" and "`Language Error`", should always be available, since interaction with many common services is based on them. The level "`Note`" is important only when annotation based services are to be used. Other standard levels are optional, "`Lexeme`" for example, but may also represent added services (for example, lexeme-bounded regular expression search and search-replace are available at this level). Pseudo-structural levels "`Character`", "`Word`", and "`Line`" are entirely optional and not very important in the standard configuration, but could be important in configurations where level-sensitive commands are more central.

Design Guideline 5 Add task-specific operand levels.

Complex operand class definitions may be needed for task-specific operand levels. In a code review, for example, a user might want to browse all uses of problematic language features, general ones such as `Goto`, and language-specific ones like C's `longjmp` and `setjmp`; these might be defined in separate levels or grouped together in a level named "Control Flow Exceptions." Even more complex and specialized levels might be useful, for example when porting software to a new compiler with guidelines such as the following:

> In the current version of GNU CC, there are three ways that the stack pointer can change value: (1) calls to `alloca`, (2) use of variable-sized objects, and (3) calls to functions with parameters that do not all fit in the argument-passing registers (e.g., more than 6 parameters). You should avoid all three in functions that call `setjmp`.

Another example might arise in the context of a language or a particular system plagued by unexpected data type conversions. A task-specific operand level might defined to identify assignments that evoke implicit coercions known to be troublesome.

The above examples might also be expressed as *Colander* constraints, adding them to level "`Language Errors`" or "`Stylistic Violations`". The designer must judge how often users may want the information; for occasional browsing, a separate operand level is the right choice.

Design Guideline 6 Add menus for groups of specialized commands.

Based on judgement about users' needs, commands can be made available on the view style's standard menu and redundantly through keybindings. Predefined menus containing suites of commands are available, for example the name-related commands (including among others `Query-Def-And-Uses`, `Goto-Declaration`, and `Announce-Type`), but only for languages in which the notion of named, typed entities makes sense. Other potential services, for example syntax-directed editing and those described in Section 5.8, would also be candidates.

Integration with Language Description Layers

References to the *Ladle* and *Colander* descriptions from the user interaction layer are usually simple, tree operator names and node properties for example, but additional support must sometimes be added to the lower layers. This is unfortunately not as simple as one would wish. In some cases automation could help, but there is always the need for at least some "glue" between the particular specifications and *Pan*'s language-independent services.

Design Guideline 7 Parameterize generalized services to the extent that they match the language.

Generalized services require a certain amount of abstraction that may or may not mesh with every language. The suite of name-related services, for example, listed in Table 4.21 is

`Query-Name-Instances`	Set current query to all uses.
`Goto-Name-Declaration`	Move cursor to declaration.
`Announce-Type-Of-Name`	Announce description of type.
`Show-Value-Of-Name-In-Place`	Show constant value if known.
`Rename`	Change name uniformly.
`Visit-Cross-Ref-View`	View sorted list of names, types, and scope names.

Table 4.21: Suite of Name-Related Commands

based on the notion of "named entity." For these to work, a view style specification must include the following information:

- Define "use of a name" in the form of a designated operand class;
- Define "declaring use of a name" similarly; and
- Implement in the *Colander* specification the generalized queries listed in Table 4.3 on page 44.

The designer faces considerable latitude in how to apply generalized models to particular languages. For this example the designer must decide which kinds of language entities can participate (procedures, labels, enumerated type constants?), and what the "declaring instance" is. At the same time, development of more detailed and flexible generalized services remains a promising line of inquiry (much of the work can be done in the extension language).

Pan's prototype syntax-directed editor is another example of a generalized service, one for which relatively more glue is necessary. As part of the user interaction specification, the designer associates a list of possible templates with each placeholder operator in the tree representation. In the *Ladle* layer the designer extends the grammar with additional rules involving placeholder operators and their textual counterparts (for example an operator `stmt_ph` with literal token "`@statement`"); this process could be semi-automated.

Visual Design

Good visual design is essential, not only for ergonomics and overall look and feel, but also to reinforce each view style's conceptual design and the system's design metaphors. The most immediately apparent choice concerns fonts, specified as a logical mapping between categories (unanalyzed text, keywords, identifiers, and comments) and specific fonts. The fonts in a map must look good together, but must also satisfy particular guidelines.

Design Guideline 8 Select fonts for keywords and identifiers to look nicely typeset; emphasize legibility of identifiers and non-alphabetic characters in the keyword font.

An analyzed document should look pleasing; proportionally spaced fonts look best. Serif fonts help contrast with unanalyzed text (design guideline 9), but sans-serif fonts have advantages too. The distinction between keywords and identifiers should not be emphasized as much as other distinctions. Legibility of identifiers is more important than keywords in general, but the keyword category unfortunately includes crucial punctuation (especially commas and arithmetic operators) that are hard to see with many display fonts. Any font assignment is a compromise in this situation, given current screen resolutions, limitation of *Pan*'s typesetting model, and the lack of fonts suitable for this use [6].

Design Guideline 9 Select a font for unanalyzed text that looks "dumb" and contrasts unpleasantly with other font choices.

Unanalyzed text must be particularly legible, even a bit more so than analyzed text, since the visual redundancy of a program known to be syntactically well-formed is not yet present. It should contrast dramatically with other fonts to help reveal the location and extent of inconsistency between text and derived data. It should look "dumb" in support of *Pan*'s "Smart versus Dumb Services" design metaphor; a good choice is a fixed-width screen font suggestive of traditional (dumb) text editors. Finally (and this usually follows from the previous suggestions) it should be visually unappealing compared to analyzed text, subtly encouraging frequent reanalysis by the user, which in turn keeps the granularity of analysis small (and fast), which in turn encourages frequent reanalysis.

Design Guideline 10 Select a font for comments that sets them apart from the real program.

Comments should be visually distinct from both analyzed and unanalyzed program text, suggesting that the designer exploit a different degree of freedom. When font style distinguishes analyzed from unanalyzed text, as in *Pan*'s sample view styles, then font size is a good choice to set comments apart. A noticeably smaller, proportionally spaced font is a good choice. Legibility is less of a problem for comments because they are primarily natural language, therefore more redundant and easy to scan. Furthermore, problems with illegible punctuation are less severe, since punctuation carries less information in comments than it does in programs.

Design Guideline 11 Define highlighters for all operand classes users may want to track continuously. Highlighted classes should also be available as levels.

Highlighters that involve standard services, for example "`Language Error`" and "`Query`", should always be defined and should be "on" by default.[20] Highlighters should be added for other useful categories, for example subsets of *Colander*'s constraints such as the class "`Stylistic Violation`". Highlighters for task-specific operand levels should in most cases be "off" by default.

Any class interesting enough for highlighting is likely to be interesting enough to navigate: a corresponding operand level should be added. The converse, adding a highlighter for every operand level, is not necessary. The user can highlight the members of any level temporarily by invoking `Query-Oplevel-Nodes`, which sets the current query result (usually highlighted) to every node at the current level.

Design Guideline 12 Define highlighters to be strict only when the information they represent would be misleading or useless during periods of inconsistency.

Strict highlighting disappears when a document becomes inconsistent; gracious highlighting persists, but may represent approximate information. Most of the time most of the information displayed by approximate highlighters is correct, so strict highlighters should be avoided. Making highlighters strict often amounts to an unwarranted interruption of service.

Design Guideline 13 Specify panel flags for important system states and for the presence of interesting operand classes. Flagged classes should also be available as operand levels.

Standard flags reveal whether a document has been modified, whether it may be modified, and whether certain services, text filling for example, are in effect.[21] A crucial flag (or flags) reveals whether text and derived data are consistent. This is important because textual clues to consistency (the special font for unanalyzed data for example, design guideline 9) are not always visible because of scrolling or when there have been only deletions.

Most other flags reveal whether any members of a particular class are present, for example the standard flag "`Language Error`". Comparable flags should be added when the mere presence of a member is important, for example the "`Placeholder`" flag, which indicates that a document is incomplete.

Any class interesting enough for highlighting is likely to be interesting enough for a corresponding flag; this additional clue is useful, since highlighting only shows class members that happen to be visible in a window.

Design Guideline 14 Flags should either vanish or change appearance when off, depending on the semantics of the information they reveal.

Flags should disappear when there is no information, when there are no occurrences of something, or when services are not in effect. Flags should dim when approximate versions of information or actions are available. For example, the flag that represents derived data dims during inconsistent periods, to suggest that the information is still there, but that it isn't as good.

[20]The background shading that reveals the extent of *Pan*'s structural cursor is implemented as a distinguished operand class with associated highlighter; this should also be "on" by default.

[21]This is the same kind of information revealed in the GNU EMACS "status line."

Design Guideline 15 Select text highlighting colors for legibility.

Color map assignments for ink and background shading must be able to draw attention. At the same time, they must reduce legibility as little as possible since the most important information is always the text. For ink, avoid highly saturated colors. For background shading, use highlighter pen pastels. Good choices, the ones that are no more dramatic than necessary, are unfortunately sensitive to variations in display hardware, ambient light levels, and individual color vision [116].

Design Guideline 16 Select alternate text highlighting colors for similarity and to support the "approximate" metaphor.

Pan's text display uses alternate color maps during inconsistency. Assign corresponding colors with similar hue so that shifts (which happen often) are barely noticed by a user attending to something else. Assign less saturation and/or luminance to the alternate choices, however, so that the current state is readily apparent and so that reduced color intensity implies a metaphorical weakening of the information.[22]

Design Guideline 17 Respect cultural color conventions.

People have strong associations with some colors. For example, use red only when it is appropriate to imply that something is wrong or to be avoided.

Design Guideline 18 Associate colors with important operand classes.

Information about particularly important operand classes may appear in several ways. Use thematic colors to connect them visually. For example, *Pan*'s sample view styles associate red thematically with language errors:

- the highlighter for class "**Language Error**" uses red ink;
- the panel flag for the class is red when on; and
- the view style logo in the alternate view for error lists is red.

Design Guideline 19 Associate graphical themes with important operand classes.

Use graphics to connect visual displays associated with an operand class and relevant keybindings. This reinforces the associations and makes keybindings easier to remember. For example *Pan*'s sample view styles associate the exclamation point thematically with language errors:

- the panel flag for class "**Language Error**" has the shape of an exclamation point when on;
- the view style logo in the alternate view for error lists has the shape of an exclamation point;

[22]Many of *Pan*'s alternate views support the same metaphor. When all information in an alternate view depends on the results of analysis, the standard ink color in the view changes to grey during inconsistency.

- the keystroke sequence "`^C !`" follows a keybinding convention to set the current operand level to "**`Language Error`**";
- the keystroke sequence "`^C ^C !`" follows a keybinding convention to toggle the highlighter for class "**`Language Error`**"; and
- the keystroke sequence "`^C ^V !`" follows a keybinding convention to visit the alternate view for error lists.

Design Guideline 20 Design expressive view style logos.

The panel logo, specified as part of each view style, is an important visual cue for identifying context. Design a logo that suggests the underlying language graphically, and create slightly modified versions to suggest task-specific view styles that share a language.

Text Editor Configuration

Some of *Pan*'s text-based facilities work best and most predictably when configured for each underlying language.

Design Guideline 21 Configure character classes and bracketing characters to agree with lexical properties of the underlying language.

The character class **`word-characters`** controls how word-oriented text commands work. Add special characters, for example hyphens and underscores, to the class when they can be part of identifiers in the language. When designing placeholder tokens for syntax-directed editing (the sample configuration in *Pan* uses tokens of the form "**`@statement`**") be sure to adjust the class accordingly (by adding '**`@`**' to **`word-characters`**).

Bracketing characters control how *Pan*'s text-based commands for moving over nested expressions; default pairs include parentheses, square brackets, and curly brackets. Add others when needed, for example angle brackets for the *Colander* language.

Design Guideline 22 Configure other text services as needed.

Options specify the default width for text windows, expressed in number of characters in the default font, and the display width for tab characters. It may be useful in some contexts to enable automatic line wrapping by default. If so, set the line-length option suitably, taking into account default window width so that wrapping usually happens comfortably before a horizontal scroll is forced. Regular expression searching is case-sensitive by default, but it should be specified case-insensitive when this is the case for the underlying language.

Bindings

Design Guideline 23 Provide both menu- and keybindings for commands; display keybindings in menu titles to help users learn them.

Many users prefer keybindings to menu bindings, but menus are effective for browsing, learning new commands, and locating infrequently used commands. A user interaction specification may request automatically created menus (for highlighter toggles and level-setting commands) with keybindings displayed.

Design Guideline 24 Assign keybindings so that "smart" commands differ from their "dumb" counterparts by an easily remembered rule.

Reinforce the "Strict versus Gracious Services" metaphor by pairing bindings to suggest that a smart command is simply a variant on its dumb counterpart. *Pan*'s default bindings follow a GNU EMACS convention by using the "`^C`" key as a mode-specific (in this case "smart" prefix). Thus, the default binding for moving forward textually is "`^F`" (command `Next-Character`) and structurally is "`^C ^F`" (command `Oplevel-Cursor-Forward`).

This guideline results partially from experience with alternate arrangements, in which some keybindings changed modes between textual and structural commands. Aside from violating the "Augmented Text Editor" metaphor (since some familiar text editing commands appeared not to work in some cases), those arrangements in effect led to unexpected behavior when familiar commands became essentially unavailable when the level was set other than at "`Character`", an interruption of fundamental services that proved unacceptable.

Design Guideline 25 Add redundant keybindings that can be efficiently typed, even if they don't follow design guideline 24.

In some cases, accelerators can be special keyboard keys whose binding is natural, for example the "`Right-Arrow`" key for `Oplevel-Cursor-Forward`, whose default binding is "`^C ^F`". In other cases, it is just a matter of convenience, for example "`Shift-Mouse-Left`" for the command `Oplevel-Cursor-To-Mouse`, whose default binding is "`^C Mouse-Left`". Bindings of the latter kind for each of the three mouse buttons permit expedient invocation of long sequences of language-based commands with the shift key held down continuously.

These guidelines and default bindings in *Pan* are far from the last word. They are based very strongly on GNU EMACS conventions, which are themselves poorly designed. Walker and Olson suggest some fruitful techniques for globally redesigning GNU EMACS keystroke bindings [133], techniques that should be equally adept at incorporating *Pan*'s design metaphors. These are discussed in more detail in Section 5.3.

Chapter 5

User Interaction in Pan

This chapter reexamines many aspects of *Pan*'s design from the perspective of the user rather than the builder. Unlike Chapter 4, which presents *Pan*'s design from a structural point of view (how the system behaves and how it is built), this chapter takes a more functional or task-oriented view (what it can do for users). The application of *Pan*'s architecture and services to users' tasks, described in this chapter, demonstrates the effectiveness of the design strategies introduced in Chapter 3 and realized in Chapter 4.

Each section in this chapter addresses a cluster of related issues, beginning with some that derive directly from *Pan*'s originally stated requirements and leading through others that arise indirectly. Each section describes the origin of the issues, along with historical context when relevant, and shows how *Pan*'s design framework enables effective solutions to design problems. A final section argues for the power and flexibility of *Pan*'s design framework by enumerating a large range of applications to which *Pan*'s framework can be applied.

5.1 Text Editing and Visual Presentation

Text editing is central in *Pan*. It is the primary medium for displaying documents to users, for displaying information *about* documents to users, and for enabling users to manipulate documents. This section examines some implications of this approach, including mention of some usability problems that arise in purely textual systems, problems that foreshadow deeper trouble when language-based interaction is added (Sections 5.2 and 5.3).

5.1.1 Text-Based Interaction

As shown in Figure 3.2 on page 23 *Pan* is designed to appear relatively conventional during ordinary text editing, in keeping with the "Augmented Text Editor" design metaphor. This reflects an early commitment in *Pan*'s design, expressed as the requirement for "familiar, unrestricted text editing", to powerful text editing services that people would gladly use. A closely associated requirement, "uninterrupted service", was to prevent *Pan*'s rich language-based features from interfering with these services. In other words, if a document appears as text than it can be treated as text.

So complete is the reliance on the textual presentation that it might not be apparent at all when language-based information is present. Rather than hide the fact completely, *Pan*'s default configuration adds the panel flag "^^" (shown in Figure 3.3 on page 24) when language-based information is being maintained. Even the text-based alternate views described in Section 4.3.11, although usually protected from change by users, permit free use of all non-destructive text commands (for example, searching, copying, and writing to files).

The use of fonts, point size, and color in *Pan* does not undercut the commitment to conventional text-based interaction because these visual attributes are independent of the text-editing interface and are not controlled directly by the user. New text created by the user typically has default attributes (standard font, no enhancements); the system changes the attributes to display information as requested by the user.

5.1.2 Typography and Document Legibility

Small experiments have suggested that high quality typography can enhance comprehension by readers of program text on paper [6,95]. *Pan*'s typographical services represent a modest attempt to achieve some of these gains. However the so-called "book paradigm" for static program publishing, described by Oman and Cook [96] and pursued in great detail by Baecker and Marcus [6], is not directly applicable because of *Pan*'s dynamic context:

- Display resolution on workstation screens is much lower than for even modestly priced printers, a disadvantage that reduces the degrees of freedom available for text display;

- More kinds of information must be displayed, for example the current state of the user's interaction with the system (cursor placement, query results, and the like), another disadvantage; and

- Many features of the display can be made dynamic and user-controllable, an advantage since not all information must be displayed at one time.

These differences lead to different tradeoffs in the static versus dynamic contexts.

For example *Pan*'s use of fonts is primitive compared to the elaborate vocabulary of program typography developed by Baecker and Marcus [6], but it helps.[1] Following established custom, automatic font shifts reveal the lexical category of program text: language keywords, identifiers, and comments. Fonts may be assigned to each category by configuration of a font map for each view style; fonts may vary in size and may be proportionally spaced. An additional font category, which has no counterpart in the static program publishing paradigms, is used for text that has not been analyzed yet (raw text that is metaphorically "dumb").[2]

Informal experience with *Pan* confirms that font shifting contributes greatly to program legibility. Optimum font assignments, however, vary among languages in different categories, for example, languages of the ALGOL family versus logic languages like *Colander*. Two kinds of visual separation are especially helpful when fonts are chosen carefully: between comments and program text, and between "raw" and analyzed text. The latter distinction supports several design metaphors and helps reveal a crucial aspect of system

[1]For a small example of typography in *Pan* see Figure 3.3 on page 24.

[2]*Pan* does not use color to distinguish lexical categories, even though Baecker and Marcus mention and demonstrate the possibility. Color in *Pan* is reserved for much more important and ephemeral use by highlighters (see the next section), keeping basic typography monochrome and visually familiar.

state: consistency (Section 5.6). Several of the design guidelines presented in Section 4.4.3 deal with these visual issues.

On the other hand, limitations of *Pan*'s approach have become evident.

- Lexical-based font shifting fails to exploit the available information; it does help users discriminate among simple categories, but it could do more.

- Experience with *Pan* confirms the observation by Baecker and Marcus that the visual properties of even well designed fonts are not entirely appropriate for programs, a problem exacerbated by inadequate pixel resolution in standard display screens. This is an especially difficult problem for punctuation characters, which can carry significant information in programs, but which are easily lost visually when proportionally spaced.

- *Pan*'s prototype text rendering engine can only display simple lines of characters; whitespace must be inserted for indentation. In the presence of proportionally spaced fonts, which have significant advantages for legibility, it is beyond the power of the prototype to perform any but the most trivial horizontal placements. The complex and fine-grained typography explored by Baecker and Marcus, which for example can vary the spacing in expressions to reveal operator precedence, and can vary the size of parentheses depending on context, requires the full power of a document quality text formatter.

With a better rendering engine, a system like *Pan* would be well equipped to produce dynamically the kind of results that the SEE visual compiler built by Baecker and Marcus achieves in a static setting. *Pan*'s advantage is the rich store of information already available about each program, as well as a general purpose inference engine capable of propagating attributes and computing complex context-sensitive information. For instance, most of the computations necessary to implement Garlan's unparsing scheme [42] would be easily expressed in *Colander*. Problems reported with the implementation of SEE, an adaptation of a traditional pass-oriented batch compiler, would be addressed easily in this information rich environment.

An experimental subsystem, the *Pan* Program Presenter, explored this prospect using presentation schema designed to exploit any kind of information in *Pan*'s run-time database, including syntax, static semantics, and directives by users [16]. In one mode of operation the Presenter performed indentation (analogous to conventional "prettyprinting," but much more flexible because of the information available); in another mode it performed even more complex layout. The Ensemble project is pursuing this approach further by removing many restrictions imposed by *Pan*'s prototype text rendering engine [48].

5.1.3 Typography and Metadata

Closely related to the value of typography for legibility is its potential for display of information *about* programs. Baecker and Marcus call this information "metadata." They tentatively identify categories (which necessarily represent only a subset of the metadata that can be important in an interactive system, where the state of the interaction can be as important as the state of the program) and explore choices for display on the page. One important concern is to avoid visual clutter that can reduce legibility.

Here the dynamic nature of *Pan* compensates somewhat for problems caused by lack of screen resolution. A *Pan* highlighter superimposes a visual effect on the text associated with program components in a designated category; for example, one standard highlighter causes linguistically malformed text to be rendered with red ink[3] and another causes sites of stylistic violations to be shaded pale yellow as if by a highlighter pen. Monochromatic analogues use stipple patterns, although much less effectively because of inadequate display screen resolution. Color, analogous to fonts, may be configured per view style by assignment to color maps for logical foreground and background categories.

The highlighter mechanism has several advantages as a channel for metadata.

- Highlighters superimpose the display of metadata onto the user's primary visual field, an approach sometimes exploited for other information intensive tasks. Pilots, for example, can benefit from the "heads up" approach that superimposes the display of data concerning on-board systems onto the more important visual field beyond the aircraft.
- Users may reduce visual clutter by switching highlighters on and off, depending on the information needed at the moment.
- The membership (highlighted text) associated with any highlighter changes as the program and context change.
- The semantics of a highlighter (the rule that determines class membership) can be arbitrarily complex and context dependent, for example syntactic categories, questionable usage, hot spots identified by profile information, and many more.

Other display features mentioned by Baecker and Marcus would also be useful for certain kinds of metadata: area shading, small glyphs such as stop signs and pointing fingers, graphical rules (lines), and others. Unfortunately none are supported by *Pan*'s prototype text rendering engine.

One aspect of *Pan*'s highlighter design has no counterpart in the static world of the book paradigm: the information that drives them can sometimes be out of date and therefore unreliable. Following the "Strict versus Gracious Services" design metaphor, highlighters can be configured to be "gracious," meaning that they continue to operate in this situation. Displaying unreliable information, however, risks misleading the user. The typographical solution in *Pan* relies on two sets of color maps, one used when information is reliable (when the document is consistent) and one when it is not (when the information is only approximate). When configured appropriately, following design guideline 16 in Section 4.4.3, the transition from exact to approximate information is accompanied by a perceptible dimming of highlighter colors, metaphorically suggesting the change, but distracting as little as possible. As elsewhere in *Pan*'s design, the user is presumed capable of reasonable judgement about the relative accuracy of this kind of approximate information, as long as the the distinction between approximate and exact information is evident.

Pan supports other, non-typographical channels through which metadata can be displayed to users: panel flags, the panel annunciator, and alternate views. Section 5.4 shows how all these can work together to provide alternate and sometimes redundant display of even a single category of information (language errors in this case).

[3]Ink colors are never so high chroma as to impede legibility.

5.1.4 Models of text structure

Designing a text editor is a surprisingly difficult undertaking. Among the many challenges is the need to exhibit a coherent conceptual model of what is being edited, a model reflected by both visual presentation and editing operations. *Pan* also presents an effective model of language-oriented editing, subject to the requirement that it not clash with the textual model (Section 5.2). Models clash when they suggest to users conflicting interpretations of how the editor is behaving or should be expected to behave.

Although seldom explicit, the design of effective conceptual models for text editors has always been a problem. For example, some of the earliest interactive editors operated on a virtual deck of punch cards; users could think of these editors as card punch machines with an erasing backspace key. Two important historical improvements added support for lines of varying length and eventually replaced the line-oriented model with a stream model. Meyrowitz and van Dam argue that the long term significance of these changes was that "displayed text was no longer considered to be a one-to-one mapping of the internal representation, but rather a tailored, more abstract view of the editable elements" [86].

The conceptual distinction between a one-dimensional text stream (the "editable elements") and a virtual two-dimensional page (the "displayed text") usually makes itself most apparent in the mysterious and idiosyncratic behavior associated with whitespace characters (spaces, newlines, and especially tabs). This behavior leads to a class of user problems best described as "ostension errors": how a user may identify an object by pointing at it, without reference to its components or to the area surrounding it [39]. For example, typical text editors permit cursor placement by pointing into the left margin but not the right margin; it is impossible for the user to "talk about" the right margin naturally when communicating with the editor, since the user is only permitted to "point at" (invisible) spaces in the text stream. This kind of behavior contributes greatly to the difficulty novices encounter when learning text editors, especially when misled by the *typewriter* and *blank page* metaphors [30,79].

This bit of historical reflection offers two lessons for the design of *Pan*. The first is that a well designed user-interface, together with user experience, can compensate for an apparently inconsistent underlying model. People become quite proficient with text editors, presumably building something like diSessa's *distributed models* [113] of the editing domain. The effect is so powerful that experienced users of text editors are often tempted to instruct novices with obviously inaccurate metaphors such as *blank page*. The second lesson is that learning to use any text editor is a difficult task, one that should be expected of users as seldom as possible. This observation lies behind the commitment in *Pan*'s design to emulate familiar text editing models.

5.1.5 Text Cursor and Selection Models

Familiar text editing models, however, vary. They also lack the generality needed for a system like *Pan*, where language-based services are modeled as extensions to familiar text services. This becomes most clear when evaluating conventional models for navigation and selection. Conceptually the two imply different kinds of information, reflecting the distinction between location and contents (a selection is usually a contiguous subsequence in the one-dimensional stream model), but in practice text editors blur them. As mentioned in Section 4.3.1, *Pan*'s model of text editing is a hybrid based on elements of both the

EMACS [122] and Macintosh [108] models, and the relationship between cursor and selection accounts for the most blatant disagreement between the two.

Macintosh Model The user may select text by pointing and dragging. Selected text becomes the operand for subsequent operations; for duration of the selection there is no separate cursor — the only locational state is the one implied by the selection. When no text is selected, a cursor appears in the text stream, positioned between two characters (or before the first character in the stream or after the last), which the user may move freely. This cursor is locational (with respect to insertion, for example) but is also used to infer an operand for commands like delete forward and delete backward, according to the rules of the particular command. Even though these two situations have different implications for location and selection (both are folded together), the two are presented to the user as a single "cursor" that can contain characters from the stream (when selected), but which often contains zero characters.

Emacs Model An EMACS selection (called a "region") is distinct from but linked to the cursor. The cursor is always present, displayed as a box positioned over a single character in the stream; to a Macintosh user it would appear as a selection of a single character. The cursor is nevertheless mostly locational, having semantics equivalent to the Macintosh cursor with no characters selected. To operate on a contiguous region of text, the EMACS user must first place an (invisible) mark at the current cursor location; from that time on the text between the mark and any cursor location is an implied selection (also invisible) that may be used as an operand (or may not, depending on the subsequent command).

***Pan* Model** A *Pan* user makes a (visible) selection with the mouse, which then becomes the operand for both the Macintosh-like operations `Cut` and `Copy` and for Emacs-like region-oriented commands.[4] Unlike both other models, the cursor and selection are separate, and the cursor can be placed elsewhere while a selection persists unchanged. Unless a user explicitly does this, however, many sequences of operations familiar to both kinds of users work much as expected. Differences among the models only show up in more complex situations, for example when making selections larger than the display window.

The *Pan* model bridges the gap between the two other models; users familiar with one or the other find no initial barriers to using *Pan* for text editing. A more important aspect of the model is that it shifts users gradually toward thinking of location (cursor) and contents (selection) as two distinct pieces of the editing context, used for different purposes. This conceptual shift helps with the layering onto the text editing model of language-based operations, as described in Section 5.3.

5.2 User Models of Document Structure

User and editor must be able to communicate about what is being edited. A central problem in the design of any interactive editor is that the structure of an object being edited seldom

[4]Unlike the Macintosh, typing text does not first delete a pending selection; although this policy could be implemented it would complicate the organization of more complex services.

maps nicely to the object's visual **presentation** created by the system. Presentation and structure are quite similar for familiar text-oriented editing, so simple kinds of navigation and pointing are obvious and effective. Even here small discrepancies create the kind of problems, mentioned in the previous section, that make ordinary text editors confusing for beginners.

For documents with rich language-based structure the problem is potentially much more troublesome; formal language structure is both less familiar to users and less evident from textual presentations. Ignoring for the moment problems that arise from the mixture of textual and structural models (Sections 5.3 and 5.6), there are two important aspects of this user model: how documents decompose into structural components and how those components are named for communication between user and system. The design of this model amounts to an exercise in user interface design, applied here to the interface between documents and users.

Pan's design framework embodies a solution that rests on several points:

- The user model of document structure is *designed* for each language-based view style, depending only loosely on the underlying language description;
- In the user model the visible text *is* the program, and structural components are regions of that text;
- Components are *named* in language- and task-specific terms; and
- Operands of structure-based commands are named instances of component classes, for example "this statement" and "next language error".

The rest of this section discusses user models of document structure in more detail.

5.2.1 The "Structural Hypothesis"

Early syntax-directed editor designs were based on the presumption that language-defined structure is an effective organizing principle for user interaction, for example: "In the [Cornell Program] Synthesizer, the fundamental tenet is that all operations are based on the underlying program structure" [126]. In this case "the underlying program structure" was explained to be an abstract syntax tree; cursor movement, for example, was presented to the user as navigation over nodes in a tree. However experience with early editors of this kind showed that users had difficulty with fine-grained structural interaction, so the hybrid approach was developed. For example, even the Cornell Program Synthesizer did not support its "fundamental tenet" at granularity finer than a statement, shifting instead to text-based interaction that relied on a parser.

Section 2.2.1 described three problems with the structural hypothesis. The first problem is that internal representations are typically designed to meet the needs of tool implementations, not people. *Pan*'s internal representation is based on an abstract tree,[5] and *Pan*'s language description mechanism offers considerable flexibility in the design of this representation. Experience with language descriptions for *Pan* confirms that the design of tree representations is driven primarily by issues concerning language description and analysis.

[5]Using incremental parsing to maintain an abstract tree representation demanded novel specification and implementation techniques [10].

The second weakness with the hypothesis is that editing operations carry additional cognitive overhead, since users must understand the complex relationship between the tree and its two-dimensional textual presentation. The third limitation is that simple syntactic categories do not correspond naturally to the kind of information users consider when thinking about programs.

The unifying theme behind these weaknesses is the observation that people understand programs in very different terms than those reflected in language-based analyzers (compilers for example) whose implementation decisions do not address the need for a coherent conceptual model for documents in the language.

5.2.2 A Separate Model for Users

In *Pan*'s design framework conceptual models of document structure are decoupled from internal representations. The view style specification mechanism described in Section 4.4 provides a loose framework in which the author of each view style is expected to design a model of document structure, following general guidelines. Each view style design is understood to be specific to a particular underlying language, but also potentially to particular groups of users performing particular tasks.

Each view style reveals to users relevant aspects of document structure, specified in terms of **operand classes**. Operand classes are implemented as arbitrary, possibly overlapping collections of nodes in the internal abstract tree representation, defined in terms of any kind of information available (for example, "declarations" or "assignment statements that evoke implicit type coercion"). These classes define potential operands for *Pan* browsing and editing commands, but need not reveal implementation decisions. For example, some classes may aggregate more than one kind of internal tree node when the distinction is judged unimportant or potentially confusing to users.

That operand classes may overlap distinguishes them from operator-phyla [63] and other grammar-based approaches to interaction; this becomes important with more complex kinds of operand classes. The operand class "`Language Error`" (Section 5.4) overlaps many of those classes; for example, a single internal structure might represent both a "`Statement`" and a "`Language Error`".

Furthermore, operand classes need not be defined for all possible internal nodes; those not defined within any operand class are essentially invisible to users. This possibility, combined with *Pan*'s incremental parser, represents an alternate solution to the "intermediate node" problem that vexes many syntax-directed editors [73].

5.2.3 Text-based Models of Structure

Operand class specifications in each *Pan* view style define user-visible structural components in terms of nodes of the internal tree representation having specified properties. Following the "Augmented Text Editor" design metaphor, and in contrast to the structural hypothesis mentioned above, the user's model of document structure in *Pan* is grounded in text. The user need not consider any embodiment of a "`Statement`", for example, other than its constituent text. From this perspective, an internal tree node is little more than a convenient place for the system to maintain information associated with regions of text; another internal representation might serve just as well.

Behind this approach is the observation that people generally think not in terms of trees but in terms of familiar structural components, the ones that might be described in an informal language description. For programs these might include procedures, declarations, statements, and expressions. People understand nesting but only weakly. For example, even though natural language permits arbitrary nesting, people find sentences with even three nested levels difficult to understand. Other naturally nested information structures, for example hierarchical file systems, are often most successfully presented to users using task-centered metaphors, for example the Macintosh folder metaphor [108].

In practice, *Pan* displays a document component to the user only by highlighting the text in some fashion, for example, when the cursor is set structurally. Following the "Heads-Up Display" design metaphor, any operation on a structural component is presented as an operation on its text. This approach solves several problems associated with the structural approach, most importantly maintaining congruence between what is seen and what is being edited.

5.2.4 Names for Components

Related to structural decomposition is the choice of terminology by which document components can be named. Simple structure editors (especially when designed to support multiple languages) may not name components at all, requiring the user to think in terms of representation-oriented commands (for example `Left`, `Right`, `In`, `Out`, and `Delete-Subtree`). A single-language editor can provide more natural language-oriented commands (for example `Next-Function`, `Previous-Declaration`, or `Delete-Statement`), but this approach doesn't generalize across languages. In some cases appropriate names overlap and depend on context; for example a structural component internally called "variable" in a programming language representation might be conceptually both a "variable" and an "expression" in some contexts but only a "variable" in others.

All operand classes in *Pan*, and therefore all structural document components about which system and user can communicate, are named in user- and task-specific terms, independent of implementation.

5.2.5 Interaction Using Operand Classes

Most structure-based user interaction in *Pan* exploits operand classes: navigation, highlighting, projection, and many panel flags are based on this mechanism. The class mechanism and terminology itself, however, is not part of the model. Instead, all interactions are wrapped in specific, named instances, for example "this statement here" (pointing with mouse), "move to the next procedure," "highlight all language errors," and "display all declared variables" (in a cross-reference).

This preference for the concrete over the abstract, for the instance over the class, addresses the tendency of people to favor the same approach. This is an instance of the more general observation that users' perceptions of systems are much more sensitive to surface representations than to underlying structure.

5.3 Mixed-mode Editing

Section 5.2 described how *Pan* users are encouraged to think about document structure, as both text and structure. In this section techniques are discussed for enabling the system to operate in both modes simultaneously, the approach being to broaden rather than narrow the user's options. The user should be able to edit textually any time, any place in the document presentation; it should be equally possible to edit structurally.

Doing so without confusing users, however, presents a challenge for user interface design (in this case between user and system). The authors of the Mentor system justified their own decision not to attempt the mixture by noting that

> A major drawback of mixing structure editing and display editing is that the user would have to learn how to use two command languages instead of one. We believe that most users would stick to either mode, but would not like mixing them [29].

Pan's design is based on the assumption that users *would* be comfortable mixing the two as long as they do *not* have to use two command languages, which is to say, as long as little mental overhead arises from the mix. The approach taken in *Pan*'s design rests on three points:

- It is meaningful to invoke any command, whether text- or structure-based, at any time; thus, there are no *modes* in the traditional sense.
- Structure-based interaction is presented not as a separate set of services, but as especially well-informed text-based services following the "Augmented Text Editor" design metaphor.
- Each structure-based operation is presented as a "smart" variation on a text-based operation, following the "Smart versus Dumb Services" metaphor; structural commands are easily remembered, and the metaphor reinforced, by careful attention to command formation.

This section describes in more detail how the two kinds of editing are combined in *Pan*, and in particular how the many opportunities for user confusion may be avoided.

5.3.1 Shifting Perspectives and Mixing Commands

A text-oriented operation followed by a structure-oriented operation implies a shift of perspective about the document, on the part of both user and system. Humans are adept at shifting perspective, and do so frequently to suit the cognitive task of the moment. For example, studies of experienced programmers reveal that both reading and authoring activities involve a variety of fine-grained cognitive tasks, with rapid switching among them [74,107]. *Pan* supports these activities by being ready to operate in either perspective at any time.

In contrast, many language-based editors that provide mixed text- and structure-oriented operations require user activity, both mental and physical, to shift the system's perspective. This can distract, slow down, and possibly confuse the user. For example, some editors require that a structural component be specially selected for text-oriented editing;

the textual content of the component then becomes the *focus* until the user wishes to resort to structure-oriented editing, at which time another explicit action may be required. Some editors provide a textual focus only in a separate window, preventing the user from editing textually in the natural visual context.

Any *Pan* editing command, text- or structure-oriented, may be invoked without prerequisite. Two mechanisms make this work. The first, automatic reanalysis (Section 4.3.9) ensures that derived information is consistent with the text before performing any operations that require it. In many editors a similar transition can be interrupted by analysis that encounters ill-formed text. The ways in which this problem is avoided in *Pan*'s design are discussed in Section 5.4.

5.3.2 Navigation

The second mechanism is the dual nature of *Pan*'s edit cursor, described in Section 4.3.7. Rather than reveal to users two separate pieces of system state that involve navigation, as do systems that have both a text cursor and structural "focus," *Pan*'s cursor is presented as a single entity whose state (and appearance) varies only slightly between "dumb" (when positioned textually at the beginning of a statement, for example) and "smart" (when positioned at the same location by a statement-oriented structural command). This design resolves the "point versus extended cursor" problem identified by Teitelbaum and Reps [126] by providing both behaviors simultaneously.

The difference between the two cursor states, as perceived by the user, is one of extra information: whether the cursor somehow "knows" that it is at a statement (because the user just instructed it to move there) or whether it doesn't know (yet) about the statement or any other structural component. The operational consequences of the two cursor states are slight; subsequent operations that need a cursor location simply use the appropriate aspect. If the cursor has no structural aspect, then one is inferred from the text location (using techniques described in Section 4.2.7) as if the user had simply pointed there with the mouse before invoking the command. This apparent sloppiness, combined with *Pan*'s operand level mechanism, addresses the ostension problem, making it possible for the user to mention "this statement" (by pointing, for example) or "the next declaration" (relative to the cursor) without regard for the state of the cursor. Users are never expected to express tree-oriented commands (e.g. `Up` or `Left`).

People have little difficulty with the ambiguity, presumably for the same reasons that they can shift perspectives themselves so effectively and can manage complex, possibly ambiguous domains [113]. For example Raskin reports that a dual-aspect text cursor solves a particular user-interface problem: his cursor simultaneously highlights a character and underlines a character, but not always the same one [100]. Each aspect identifies the operand of a subsequent command, and careful design produced a style of interaction that reduced both learning time and error frequency for experienced users. The crucial point is (as with *Pan*) that both aspects are visible and predictable, an example of Norman's *mode feedback* [91].

Navigation commands that operate either structurally or textually are listed in Table 5.1. The two searching commands operate textually and lexically, but have not yet been implemented structurally. `Cursor-In`, `Cursor-Out` are additionally supported for compatibility with certain kinds of structural editors, even though they have no natural counterpart

```
Cursor-To-Mouse
Cursor-Forward
Cursor-Backward
Cursor-To-First
Cursor-To-Last
Cursor-Search-Forward
Cursor-Search-Backward
```

Table 5.1: Mixed-Mode Navigation Commands

in text editing.

5.3.3 Command Formation

It is important for any interactive system that the user be able to invoke desired operations without unnecessary mental overhead. As one step in the process, users must syntactically form and execute a command in the input language of the system; for *Pan* and similar systems this amounts to assembly of a sequence of keystrokes, mouse button presses, and menu selections. As with many systems designed for expert users, mouse-invoked menus in *Pan* play a specialized role: they make simple commands available to users with little prior learning, and they help teach users about available operations. Command formation by menu is therefore assumed to take place in a context of high cognitive overhead; the relative slowness of menu-based interaction is not an issue.

For commands invoked frequently by experienced users, however, efficient command formation is important. Two issues affect this phase of interaction:

- Difficulty deciding what sequence of keystrokes will invoke a desired command; and
- The time it takes to press them.

Walker and Olson argue convincingly that the former is more important than the latter for systems like *Pan*: "the time involved in making and correcting a keybinding that is difficult to remember is far more costly than the extra keystroke that is required in an easily remembered keybinding" [133]. Given that *Pan*'s command set and default keybindings are presented to users as an augmentation of the already complex GNU EMACS conventions, the paramount challenge is to make mixed-mode commands both memorable and easy to infer. Experience with *Pan* reveals a further dimension: the general scheme adopted for remembering keybindings can either support or detract from important design metaphors.

The general approach taken in *Pan*'s design to chunk together analogous textual and structural commands and then to provide easily remembered techniques for refining the semantics of basic command invocations. The potential complexity involved can be seen by considering the user's options when invoking the simplest possible command, to move the cursor forward:

1. The user may want to move forward either textually or structurally.

2. When moving textually, the user may want to move using familiar text-oriented commands, for example by a character, a word, or a line.

3. When moving structurally, the user may want to move to a component called a **"Statement"**, a **"Procedure"**, or any other class made available by the author of the view style.

4. The user may want to move more than one at a time.

Each of these complications is addressed in turn. The general solution, as with much of *Pan*'s interaction, is to emulate familiar text editing models and require as little additional learning as possible. For *Pan* the decision was made to embed complex command formation into the GNU EMACS model, but the same kind of adaptation could be applied to other models as well.[6]

First, the text-oriented command for moving the cursor forward in *Pan* is named **Next-Character** and is bound by default, following GNU EMACS convention, to the key "^F". Its structural counterpart for moving forward is implemented as a separate command, named **Oplevel-Cursor-Forward**, but the *Pan* model dictates that it be presented to the user as a simple variation of the command bound to "^F". Since the standard GNU EMACS command modifier "^U" is already used for command multiplers (see below), some other modifier must be used. The alternative (a completely unrelated set of keybindings for structural commands) would result in unmemorable bindings that did nothing to support the metaphorical relationship between the two kinds of commands in *Pan*. Another GNU EMACS convention sets aside the key "^C" as a prefix key for special modes, and this permits adoption of the following rule of thumb in *Pan*:

> To invoke the smart version of an ordinary textual command, precede its invocation with the "^C" key.[7]

Thus, **Oplevel-Cursor-Forward** is bound by default to "^C ^F". This rule of thumb is easily remembered and even applies to mouse buttons. For frequently used commands so-called accelerators are also in place, menu and keybindings that are much faster to invoke, but their use is optional; for example, the right arrow key on most keyboards is also bound by default to **Oplevel-Cursor-Forward** for quick single-step navigation.

Second, following GNU EMACS conventions, standard variants on text-oriented navigation commands are bound to separate keystroke sequences: **"Escape F"** is bound to **Next-Word** for example. Also like GNU EMACS, word-oriented commands in *Pan* are configured by character classes that define, for example, which characters should be considered parts of words and which should be considered word separators. Character classes may be adjusted as part of each view style so that this kind of text-oriented command operates in a natural way for the underlying language. Note that *Pan* also supports lexeme-oriented commands that use structural rather than textual information, and in many cases this can be more exact than class-based text commands. Both are necessary, however, since they

[6]For example, Walker and Olson present a rationalized keybinding set for EMACS editing commands and demonstrate that it is easier to learn and remember; *Pan*'s augmentation would fit nicely with that set too.

[7]This rule is very similar to the hierarchical approach to designing the keybindings advocated by Walker and Olson.

operate differently in some cases; a comment is a single lexeme, for example, but contains many textual components.

Third, an entirely new mechanism in *Pan* permits users to specify the granularity of structural operations, for example whether to move forward by a "`Statement`" or a "`Procedure`". The operand level mechanism, described in Section 4.3.6, helps users cope with the generality of many languages, many view styles, and many potential operand classes; it is a weak input mode that serves as an alternative to the kind of command profusion that might otherwise be needed. The user specifies a current operand level in each window by menu selection (or keybinding), where the options listed in the menu are determined by the view style specification. The current level modulates structural commands such as `Oplevel-Cursor-Forward` but has other effects and implies no restrictions on user actions.

Finally, *Pan* supports command multipliers following GNU EMACS convention for use of the "`^U`" prefix. The default multiplier for commands is one, but the prefix may be used to specify any number of repeats, for example, to move forward by three statements.

Informal evaluation of several earlier keybinding sets led to this solution. Given the ease with which alternate keybindings can be implemented in a customizable system like *Pan* and EMACS (all but the most naive users can learn how to rebind keys), it is surprising the degree to which variations colored users' experience with the system. One early arrangement, in which simple keys such as "`^F`" were level-sensitive, led many users to perceive the entire system as much more "moded," and therefore "unusable" than it needed to be; this was a failure to meet the requirement for "familiar, unrestricted text editing", since users felt that text commands were sometimes unavailable or produced unpredictable results. A redesign of the default keybindings, based on more careful consideration of basic design metaphors, corrected the problem and led to an improved perception of the system by users.

This experience with keybindings in *Pan* mirrors a result noted by Ledgard, Singer, and Whiteside during a carefully controlled experiment: subjects in an evaluation of command languages for text editors failed entirely to detect the equivalence of two editors that differed only in details of their command languages [72]. As in many other contexts, surface manifestation (the syntax as opposed to the semantics) *is* the system for most users most of the time.

5.3.4 Structural Selection

Unlike navigation in *Pan*, where the structure cursor is presented to users as an extension to the text cursor, the need for a structural analogue to text selection was not anticipated. Although navigation and selection are distinct in the text editing model, they are blurred together in the extended language-based model. This section describes the limitations of the current model and discusses a proposed model for structural selection that addresses them.

In *Pan*'s current configuration, the text selection aligns automatically with the text of the cursor whenever set structurally. This expedient allows a structure movement to be combined efficiently with a subsequent textual `Copy` or `Delete` operation, both of which operate on the text selection.

Experience has shown this arrangement to be inadequate, just as would be the case if it were possible for user and system to communicate textually only about a single character at a time. User and system must sometimes communicate in terms of multiple structural components simultaneously. For example, the query mechanisms described in Section 4.3.8 must return an answer to the user in just this form (an arbitrary collection of structural components) and a special-purpose mechanism now plays this role. Further, several applications designed for *Pan* require the user to designate more than one structural component as collective arguments to a single command; these are unimplemented because there is at present no support for multiple operand selection.

The need for collection-based interaction arises in a surprising variety of contexts. For example, a recently developed compiler-debugger interface permits stepping through optimized assembly code; highlighting the corresponding source code at each step sometimes results in disjoint components being highlighted because of optimizations [19]. Systems that deal with higher-level programming constructs, including many transformation based tools (for example the KBEmacs editor, in which clichés manifest themselves visibly as non-contiguous components [135]), likewise need this kind of interaction (although Waters mentions that the "EMACS-style program editor," with which KBEmacs is integrated, can't even support this).

Task analysis has led to an unimplemented design that generalizes selection in precisely the same way that the cursor has been generalized in *Pan*'s model. In this model there would be only one notion of selection, which might or might not be set at any given time (in contrast to the cursor, one of which is present in every window). The crucial generalization at the textual level would be to allow *disjoint* regions to be selected, that is, the current selection would be modeled as a *set* of text regions. The potential complexity of this generalization when seen from a text-only perspective does not justify its addition, since complex policy questions would arise concerning overlapping and nested selection components. In practice, this selection would never contain more than one member during normal text editing, and all text-oriented commands would work as expected.

The selection could, however, become "smart" in precisely the same way that the cursor does: by alignment to correspond to structural components. *Pan*'s query mechanism, instead of invoking a special-purpose highlighting mechanism, would then simply "select" the result. This would enable the query mechanism to be presented to users as a "smart" version of the familiar text-based query mechanism based on regular expressions. Each component of the answer would be selected textually (displayed by underlining) and the text would additionally be highlighted in some distinguished way to show that the selection is structurally motivated. Most cases of multi-selection would result, as in this example, from invocation of suitable commands; additional user commands would seldom be necessary.

There would be commands, however, for the user to construct a multiple selection manually, for example with commands to select the component at the cursor, to add the component at the cursor to the selection, to delete it, and to navigate structurally over the current selection (much as the user can now navigate over members of the most recent query result).

Underneath all this would be a general mechanism for collecting, naming, and retaining arbitrary collections of nodes on the fly. This would be available to advanced applications of the sort mentioned above.

5.3.5 Simple Editing

The prototype implementation supports no user commands that modify internal document structure directly. A **Delete** command, invoked with a structural selection, removes the text associated with the selected component. The internal structure corresponding to the deleted component persists until the next reanalysis, but it is invisible to the user: automatic reanalysis will delete it before any commands can use it and in particular before the cursor can land on it. **Cut** places text in the clipboard, and **Paste** simply inserts text from the clipboard. If the context is appropriate, subsequent incremental analysis derives the equivalent structural information quickly.

This implementation costs a small amount of analysis time by discarding derived information when the user moves structural components. On the other hand, it guarantees the integrity and well-formedness of the document's internal representation, since the language definition is already built into *Pan*'s parser.

Complex mechanisms for direct structural editing can be a source of confusion to the user, since those editing operations may fail, something *Pan* commands never do. Worse, they may fail for the kinds of reasons best hidden from the user, for example the presence of "intermediate nodes."[8] Solutions ultimately involve guessing about the user's intent, an approach avoided in the design of *Pan*. When a structurally inspired **Cut** and **Paste** sequence in *Pan* violates the underlying language definition, the operations succeed anyway and the problem is diagnosed by precisely the same mechanism that handles other language violations (discussed in Section 5.4).

The only loss at present caused by this text-based implementation is the loss of non-derivable annotations on document components during **Cut** and **Paste** sequences. Strategies are possible within *Pan* that avoid this information loss and provide functionality equivalent to direct structural operations, but the successor to *Pan* is addressing the problem in more fundamental ways.

5.4 Ill-Formed Documents

Inherent in the syntax-recognizing approach adopted for *Pan* is the certainty that documents being modified are more often than not **ill-formed**: at variance with an underlying language definition. This section discusses the problem of ill-formed documents from two perspectives. The first is the need to meet *Pan*'s requirement for "uninterrupted service", using a pervasive set of techniques organized around the "Imperfect World" design metaphor to make all of *Pan* work as well as possible in this situation. In the second perspective, a cluster of *Pan* services[9] make diagnostic information available to users in a variety of ways, demonstrating the flexibility of the basic elements of user interaction detailed in Section 4.3; the presentation of diagnostics is an example of a *Pan* application for exploiting one kind of metainformation about documents.

Ill-formed documents are often said to contain "errors," a pejorative term reflecting the limitations of many analysis methods. Many language-based editors that permit text-oriented editing inherit this bias. Unable to analyze ill-formed documents, these editors

[8] Section 2.2.3 presents an example of a structural cut and paste sequence that might fail this way.

[9] A cluster of low-level services and configurations in *Pan* that exploit a particular kind of information or relate to a particular user task is sometimes called a *Pan* **application**.

insist that the user correct any newly introduced "errors" before proceeding. Often claimed to be an advantage, because it limits the extent and duration of "errors," 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. An "error" may often 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 may simply be unable to understand what the user is doing [76].

Pan's approach to the treatment of ill-formed documents pervades the system. It is an entirely normal state in *Pan* for documents to be ill-formed; every attempt is made to provide all services in the presence of any such **variances** with respect to the constraints imposed by the language description. In fact, information about variances is an important and useful kind of derived information, available whenever the user wants it.

Pan's approach decouples ill-formedness from inconsistency between the textual and language-based aspects of a document, discussed in Section 5.6. A consistent document may or may not be well-formed; likewise an inconsistent document may or may not be well-formed, but the system can't know in this case.

5.4.1 Tolerance for Variance

Pan's design includes two special mechanisms for handling variances, reflecting the two layers in which the author of a *Pan* language description defines well-formedness: a context-free grammar and contextual constraints. These layers are reflected in turn by separate analysis techniques, reflecting the traditional division of analysis in language-processors into syntactic and static-semantic analysis.

As described in Section 4.1.2, *Ladle* builds internal tree representations as specified by the relevant part of each language's description. *Ladle* implicitly extends this specification to include special document components, created automatically during incremental syntax analysis to represent instances of **syntactic variance**. Each of these special components is named after a specific kind of variance, for example "malformed statement". These components may retain well-formed subcomponents produced during prior analysis. For example, a "malformed block" might still contain well-formed statements that are accessible to the user as instances of "`Statement`". As much derived information as possible is retained along with these subcomponents for later analysis. This is an important implementation strategy for bounding the effects of minor or ephemeral syntactic variances.

As described in Section 4.1.3, *Colander*'s semantic analyzer attempts to prove that every description-defined constraint on each document component is satisfied.[10] For example, many programming languages require that all variables be declared. In a *Pan* language description, this requirement is enforced by placing an appropriate constraint on each document component where a variable can be used. If an undeclared variable appears, a

[10]A *Pan* language description may include useful additional constraints that do not derive from the language definition, but rather from local and possibly personal conventions for the use of each language.

constraint variance is added to the relevant component as one of many properties that may be recorded there.

5.4.2 Classes of variance

As with most language-based systems, the distinction between syntactic analysis and contextual constraint checking is fundamental to *Pan*'s language description and analysis infrastructure. Early versions of *Pan* inappropriately exposed this distinction to the user in the form of the two predefined operand levels that corresponded to the underlying implementation: "**Syntax Error**" and "**Unsatisfied Constraint**".

A more general mechanism now permits the author of each language-based view style to specify one or more classes, each of which may contain arbitrary subsets of possible variances. For example, a simple view style might reflect the way naive users think about errors; the single operand class "**Language Error**"[11] would include all possible instances of syntactic errors and unsatisfied constraints. On the other hand, experienced users seem to place variances into categories according to various criteria, for example severity, nonlocality, or perhaps even level of surprise; this may likewise be modeled by appropriately defined and named operand classes. This generality becomes even more important when extra-lingual constraints are added, since they are implemented as *Colander* constraints, but reflect a different category of metainformation from the perspective of users.

5.4.3 Diagnostic Services

Well-formed documents in *Pan* differ from ill-formed documents only by the absence of variances, and both text- and language-based operations may proceed in the presence of variances. *Pan*'s diagnostic application offers the user several different ways to communicate *about* variances, reflecting a rich set of channels available for meta-information. As with most of *Pan*'s interface, all are optional and under user control. Furthermore, the implementation of these services serve as an example of how visual design, in particular design guidelines 18 and 19 described in Section 4.4.3, can connect different channels for a particular class of meta-information. In the default configuration (shown in in Figure 3.6 on page 28) the color theme for language errors is red and the graphical theme is the exclamation point.

Announcement As part of each incremental analysis, a panel message notes the number of variances present.

Panel Flags Panel flags appear when specified kinds of variance are present; the default flag for class "**Language Error**" appears as a red exclamation point. This may be all the information a user needs when it appears subsequent to a small modification.

Highlighting Highlighters render text associated with specified kinds of variances specially; the default highlighter for class "**Language Error**" uses red ink. Experienced programmers often diagnose problems at a glance, once attention is drawn to their location.

[11] Although these are "variances" in *Pan*'s design, users still know them by the conventional term.

Navigation The user may request more details setting the level appropriately (default configurations make the level "`Language Error`" available) and navigating structurally. The default configuration for this level includes an after-daemon that announces the diagnostic each time the cursor is positioned structurally at the level, "malformed statement" for example.[12]

Alternate Views The Error List view style, described in Section 4.3.11, lists all diagnostic messages in a separate window whose view style logo is a red exclamation point. Exploiting shared structural navigation, a simple command to set the cursor in the list causes the cursor in the primary view to move to the location of the variance.

5.5 Incomplete Documents

Section 5.4 began with the premise that documents being modified are more often than not ill-formed, and it described efforts in *Pan*'s design to provide as much service to users as possible in this situation, following the requirement for "uninterrupted service". This level of service can be improved (and new services added) through attention to the special case that arises when ill-formedness results from the absence of required structural components, for example from a newly entered `while` loop whose termination condition is still missing.

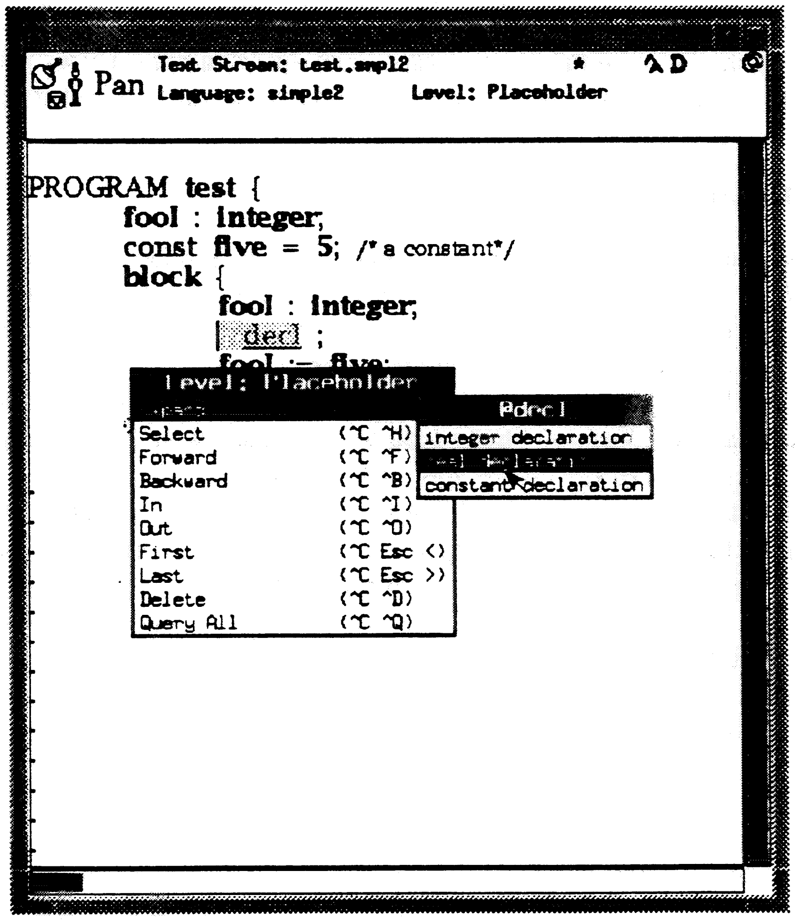
One useful outgrowth of the syntax-directed approach to editing is first-class treatment of documents that are incomplete in this way. The standard technique (prompted originally by the need to make sense of unexpanded nonterminals in grammar-based derivation trees) is to support insertion of visible **placeholders** that act as surrogates for designated classes of structural components (typically operator phyla such as "statement"), but which lack any further detail. Analyzers treat placeholders specially by not raising unnecessary error conditions in their presence. Placeholders in syntax-directed editors are typically protected against text-oriented modification; they serve instead as expansion points where "elaboration" or "expansion" commands replace them with more detailed templates that may themselves contain placeholders (a "while loop" template might replace a "statement" placeholder for example).

A *Pan* experiment with this style of interaction demonstrated that the advantages of placeholders can be made available entirely within *Pan*'s design framework,[13] and that they can in fact be implemented as just another application built upon kernel mechanisms and configured with basic elements of user interaction.

Consistent with the text-based approach to document structure that pervades *Pan*'s design framework (and unlike conventional syntax-directed editors), placeholders in *Pan* are simply text and can always be manipulated as such. For example, the string "`@decl`" appearing in Figure 5.1 is simultaneously text, a member of the language-specific class "`Declaration`", and a member of the special class "`Placeholder`". All standard structure-based commands work appropriately at both levels, and some special ones are configured to work at the "`Placeholder`" level. In particular, the level-sensitive (or "smart") menu

[12]A "malformed statement" is in both the "`Statement`" and "`Language Error`" operand levels; the announcement appears only when the user navigates at the latter level.

[13]This experiment covered only the interaction mechanisms and configuration. *Ladle* descriptions were extended manually in ways that could be automated. *Colander* support for placeholders is likewise absent, but can be implemented by manual additions to the description.


Text Stream: test.smpl2
Language: simple2
Level: Placeholder

```
PROGRAM test {
        fool : integer;
        const five = 5; /* a constant*/
        block {
                fool : integer;
                decl ;
                fool := five;
```

Figure 5.1: Using *Pan*: Expanding a Placeholder

command `Oplevel-Menu`, bound by default to "`^C Mouse-Right`", is configured in this case to be context sensitive: it presents the user with the usual menu of level-sensitive commands augmented by a submenu of templates (visible in Figure 5.1) appropriate for replacing the placeholder at the cursor.

Pan's other user interaction techniques contribute as well. A standard flag may be added to the panel, as shown in Figure 5.1, that appears when there are placeholders present. This flag (whose default appearance is a purple "`@`" to support the visual theme for syntax-directed editing) indicates that a document is incomplete in the sense being discussed here. A standard highlighter for placeholders may be installed, helping the user locate them at a glance (this highlighter is of course configured by default to color placeholder text in purple).

This approach permits a convenient mix of text-oriented and syntax-directed expansions. For example, once a user sets the level to "`Placeholder`", repeated template expansions may be invoked by iteration over the following sequence of actions:

1. The user points at a placeholder and presses "`Shift-Mouse-Left`"[14] to set the cursor, causing the placeholder to be highlighted and selected textually.

2. The user presses "`Shift-Mouse-Right`" to bring up a context-sensitive menu that includes a submenu of template expansions for the placeholder at the cursor (Figure 5.1). Selecting one invokes the expansion, followed by an automatic reanalysis.

An expansion also leaves the cursor positioned at the first placeholder, if any, in the inserted template; the user may then expand that placeholder in turn with only one additional action. An experienced user would repeat this sequence quickly by holding down the shift key for the duration and invoking commands with mouse buttons only.

A common digression from this purely structural sequence is textual replacement of the placeholder at the cursor, especially for placeholders. This sequence can be fast too:

1. The user types "`^W`" to delete the (already selected) text of the placeholder (the characters "`@identifier`").

2. The user types replacement text.

Unlike placeholder expansion, this sequence leaves the analysis slightly out of date, but the user's next structural move to a placeholder (either by pointing or invoking `Oplevel-Next`) triggers reanalysis (fast in this case since the amount of analysis is kept small by frequent analysis). Other textual digressions are just as convenient, for example moving to the end of the current line, adding a comment, and then moving forward structurally to the next placeholder.

Finally, there is no reason to limit template expansion services in *Pan* to simple expansion of nonterminals in abstract grammars. For example simple clichés of the kind supported by KBEmacs might be inserted (though not necessarily stored in any way to permit their recovery) [135]. Some might be parameterized in complex ways, for example by selection of a program fragment to fill a central "slot" in the cliché; some might need multiple structural selection of the sort advocated in Section 5.3. This all works because template expansion

[14]This keybinding is an *accelerator*, bound for convenience to `Oplevel-Cursor-To-Mouse`, a command whose memorable (and slower) default keybinding uses the `control-C` prefix instead of the shift key.

in *Pan* is implemented on top of kernel mechanisms and language-based infrastructure, not integral to them as with earlier generation syntax-directed editors.

5.6 Incrementality and Inconsistency

Any situation where one kind of information is derived dynamically from another invites **inconsistency**[15] between the two. The syntax-recognizing approach adopted for *Pan*, where language-based information may be derived from document text, is no exception. During text-oriented editing, language-based information maintained by the system sometimes disagrees with what the user sees. For example, *Pan*'s lexical font shifts become incorrect immediately after a statement has been transformed textually into a comment, and no derived information may be available at all for newly entered text.

Inconsistency between text and derived information should not be confused with a related but distinct issue: the well-formedness of a document with respect to its underlying language. Many language-based editors that permit text-oriented editing are able to resolve inconsistency only for well-formed documents: a user who modifies a document textually is not permitted to continue until all changes meet the system's requirements for syntactic well-formedness. *Pan*'s analyzers never fail (Section 4.1), and there is no need for this restriction. It is possible for a document's text to be linguistically well-formed but inconsistent with respect to derived information, for example when the user has just corrected a language error textually. It is equally possible for the document to be ill-formed but consistent, in which case the derived data includes accurate diagnostics. Section 5.4 discusses well-formedness in more detail.

Two general aspects of system behavior concern the problem of inconsistency. The first is how often and under what circumstances the system should reanalyze, given that the mechanisms involved have the potential to confuse the user, to degrade performance, and to make the system's behavior unpredictable. The second is how the system should behave during periods of inconsistency, given that services based on incomplete and unreliable information, when they can work at all, may mislead users. Workable solutions demand delicate compromises involving user interface issues, analysis methods, and system performance.

In *Pan*'s design framework these system issues are refined into four user-centered goals:

1. Don't let analysis obstruct or distract users.

2. Keep the extent of inconsistency small by analyzing often.

3. Make inconsistency visible.

4. Minimize the distinction in service.

Unfortunately these goals conflict. For example, a policy of reanalysis only upon explicit request is nonintrusive, but invites extensive inconsistency. Likewise, making the distinction between the two states visible implies a perceptible threshold, but such a threshold

[15]The term "consistency" is unfortunately used in a variety of ways not directly connected to the meaning intended here. For example, it is sometimes used to name a desirable property of a system's user interface, although it is debatable whether this meaning is either well-defined or useful [49]. Minör uses the term in yet a different way when describing his language-based editor, concerning grammars in trees in this case [87]

can distract when the state changes (for example, after the insertion of a single character immediately following analysis).

Earlier syntax-directed editing systems implicitly addressed subsets of these goals, but they did so by imposing restrictions on users and by not attempting to meet all of them. For example, a pure structure-editor prohibits inconsistency by restricting user actions to syntactically legal transformations on a derivation tree. Hybrid editors bound inconsistency to a structurally-defined "focus" and make it visible, but they impose awkward mode changes on the user, restrict options in the presence of language errors, and may often make the focus much larger than necessary.

The remainder of this section discusses interrelated policies and mechanisms in *Pan*'s design framework that address these goals. These solutions demonstrate that, when the goals are addressed together, workable compromises can be made that are consistent with *Pan*'s design requirements. Fundamental to these solutions is the following assumption:

> An experienced user is almost always a better judge of the extent and implications of recently introduced inconsistency than the system.

After all, the user is responsible for the changes and (mostly) understands them; the system seldom knows what the user is *really* doing.

5.6.1 Unobtrusive Analysis

One way to keep analysis unobtrusive is to make it fast. Pervasive reliance on incremental analysis (Section 4.1) helps. For many simple textual changes, analysis time is roughly proportional to the extent of the changes. Unfortunately, non-local linguistic structure means that the potential for extensive reanalysis is always present. A reasonable assumption here is that experienced users develop an intuition about the kinds of changes that have extensive implications and that they do not perceive variations in analysis time as caprice on the part of the system.

Even well designed incremental analyzers can incur perceptible delays on present-day workstations, so a careful policy is important. An overly ambitious policy, attempting reanalysis after every character insertion or deletion, encounters serious problems. First, analysis at this granularity would find documents nearly always ill-formed. There is little to be gained by insisting that the system perform useful analysis on documents that are in intentionally meaningless states. Second, this drain of resources degrades performance in ways beyond the user's control. Third, over-eager update of the display is visually distracting. The Magpie editor [111] successfully analyzes after every keystroke, but it supports very few services compared to systems like *Pan*.

Two somewhat less ambitious policies are possible. First, the system might *guess* when the document is reasonably well-formed and suitable for analysis. This policy shares the disadvantages of the more ambitious policy—the potential for unpredictable and uncontrollable behavior. Second, the system might restrict text editing to a bounded context (or *focus*) based on some internal structure, performing analysis automatically when the user attempts to leave the context in some way. This policy can create confusion about the exact nature of the context and is incompatible with *Pan*'s general insistence on unrestricted text editing with no overhead on shifts of perspective.

Pan's policy is lazy and predictable, based upon the assumption that the user understands the general state of the document and can judge the tradeoffs involved. Incremental analysis is only performed when requested by the user, either *implicitly* by invoking an operation that triggers automatic reanalysis (Section 4.3.9) or *explicitly* by invoking **Analyze-Changes**.

5.6.2 Frequent Analysis

Nothing in *Pan*'s analysis policy prevents a user from typing an entire document without once invoking the analyzers. This largely defeats the purpose of many *Pan* mechanisms, overtaxes other techniques for dealing with inconsistency, and renders analysis time relatively more obtrusive. Although the user is always the final authority in *Pan*'s design, several techniques encourage frequent analysis.

The overt technique is to deliver helpful applications so that users perceives analysis time as cost-effective. Since invoking strict services causes automatic reanalysis, the more frequently these are used the smaller the increments of analysis. Gracious services do not require consistency, but users will often want to invoke analysis explicitly if services are known to work better as a result.

The covert technique exploits one of *Pan*'s gracious services, lexical font shifting. The font category reserved for unanalyzed text helps the user judge the extent of inconsistency (see below), but it also disturbs the visual harmony of the display during inconsistency, subliminally encouraging frequent analysis to make it more aesthetically pleasing. This only works with appropriate font assignments, as suggested by design guideline 8, which encourages fonts for analyzed ("smart") text that look "nicely typeset," and by design guideline 9, which recommends a font for unanalyzed ("dumb") text that not only looks dumb, thereby reinforcing an important metaphor, but which also "contrasts unpleasantly with other font choices."

5.6.3 Visibility and Soft Thresholds

Inconsistency can have important consequences, so it is important that the user know when this is the case; this is another example of Norman's recommended *mode feedback* [91]. It is also important to give the user whatever help possible in judging the extent of any inconsistency. The cues must be be subtle, however, so that they do not intrude when the user is occupied with tasks for which the information is irrelevant. As a special case, transitions between the two states should be perceptible but not distracting.

It is especially important to avoid threshold effects, distracting visual changes perceived as disproportionate to recent user actions. But a single keystroke in *Pan* can trigger the transition from consistent to inconsistent. The problem is made worse by the fact that this keystroke, the first after an analysis, is nearly always the first of a planned sequence for which the transition is cognitively irrelevant; the user, having made the plan, has already made a commitment to the document becoming inconsistent. In early versions of *Pan* that first keystroke caused all highlighters to be turned off, a severe annoyance to any user making changes by following a highlighted trail through the code. Highlighters no longer do this (see below) and several other techniques help soften the transitions.

The only authoritative indicator of inconsistency is the panel flag "^^"; this flag, otherwise displayed boldface in dark green, appears dim red during periods of inconsistency. Other visible transitions include color changes for any highlighters configured to be gracious; here careless color choices can produce unpleasant visual thresholds (see below). The extent of inconsistency is revealed in some cases by the amount of raw, unanalyzed text visible, but this cue is not completely reliable: raw text may be present in areas not visible in any window, and there are no visual cues at sites where text has been recently deleted.

It is an open question whether inconsistency might be quantified usefully; if so, revealing inconsistency as a finer-grained change could be helpful to users.

5.6.4 Approximate versus Exact Services

A deeper approach to softening the threshold between consistent and inconsistent states is to make it less important to the user, meaning that the system should behave as much alike in the two states as possible. Maintaining service during inconsistency addresses *Pan*'s fundamental requirement for "uninterrupted service", but risks misleading the user by using obsolete derived data. Presumably because of this risk, other editing systems, and early versions of *Pan* as well, offer no general services that continue to operate in *approximate* modes during inconsistency.

Pan's design framework addresses this prospect with the distinction expressed in the "Strict versus Gracious Services" design metaphor, introduced in Section 4.2.3. A gracious[16] service in *Pan* ideally:

- Operates normally when a document is consistent, using exact information;
- Approximates normal operation otherwise;
- Contributes no distracting behavior to transitions between consistency and inconsistency;
- Operates correctly most of the time when operating in the presence of small inconsistency; and
- Never lies in ways incomprehensible to users.

Notions like "approximate" and "most of the time" are informal but precise, having meaning not in formal language theory but in the cognitive experience of users. By analogy a spelling checker always operates in approximate mode, but well designed ones are both unobtrusive and useful.

Lexer-based font shifting was a gracious service in *Pan* before the concept had been articulated in the design framework. Operationally, the lexer assigns font categories to all text touched during analysis, and those font assignments persist until touched again by the lexer. Applying the "strict" policy (implicitly in effect for all other services at that time) would have meant resetting all fonts to the raw category whenever a document became inconsistent, since the information upon which they were based is suspect. Actually, language theory suggests that only text to the right of any change need be reset this way,

[16]The term derives from the engineering term "graceful degradation," referring to products and services that fail gradually rather than abruptly.

but the visual effects (and loss of almost-correct information) would be just as unpleasant if not more so.

Pan highlighters began as strict services but may now be configured to operate in either mode; design guideline 12 suggests that the choice be made by considering whether approximate operation could be either misleading or useless. To a certain extent this amounts to determining whether local textual changes can have confusing and unpredictable implications for the information being displayed, but there hasn't been enough experience with gracious highlighters to articulate any more precise guidelines.

It is important that colors be assigned carefully to any highlighter configured to work graciously: design guideline 16 suggests that the two relevant colors (one for exact, one for approximate operation) support the "approximate" metaphor by having the same hue, for example, but with a bit less saturation and/or luminance for approximate mode. This avoids distracting thresholds when the state changes, helps makes the state visible, and has the correct connotation for the "quality" of the information being displayed.

Structural navigation in *Pan* remains strict but is a good candidate for elevation to graciousness. *Pan*'s internal tree representation and its connection into the text stream would permit straightforward implementation, but interesting policy questions concerning fine points and boundary questions would have to be explored; these would amount to elaboration of the "location mapping" mechanisms and heuristics described in Section 4.2.7. For example, a textual location in a new region of text would map into nearby structure using simple rules; similarly, a policy would specify a threshold of damage permitted to the textual yield of a tree node to permit placement of the structure cursor on it. As with all gracious services in *Pan*, approximate operation would be an optional convenience: the user can make all services exact at any time with an invocation of `Analyze-Changes`.

5.7 Alternate Views

As noted in Chapter 2, software documents have rich internal structure, with different relationships important to users at different times. In his 1979 essay *Beyond Programming* Winograd argued that multiple, dynamic views of software are important: "The programmer needs to be able to reorganize the information dynamically, looking from one view and then another, going from great generality down to specific detail, and maneuvering around in the space of descriptions to view the interconnections" [139].

Pan's design framework for alternate viewing, described in Section 4.2.8, represents a step in that direction within *Pan*'s current implementation framework. Simple view frameworks can display batches of derived text, permit access to textual annotations on structural components, project component classes onto textual lists, and paint graphical trees with labeled nodes. Many sample views have been implemented within these frameworks; for example tree-to-list projections include a list of error diagnostics, a cross reference of names, a table of contents, and a summary of textual annotations. As with *Pan*'s other elements of user interaction, alternate views will be as useful as the kinds of information used to drive them; the following section describes many possibilities.

Other viewing frameworks would help, giving the view designer the power to filter and transform information in more general ways. Some of these also appear in the following section.

5.8 Other Applications

The ultimate advantage of language-based interaction lies in a rich and open-ended collection of services (organized into thematic or task-related applications) that draw upon a rich repository of information to assist users in commonly performed tasks. This chapter has already described some of these, for example the collection of services that deals with the location and diagnosis of variances (Section 5.4). This final section lists other examples in three categories: those already implemented as prototypes, those unimplemented but straightforward, and those requiring further development or integration with other tools.

In all cases it is emphasized first that applications are implemented by combining generic and language-specific services, and second that the user need not be aware of the complex internal representations needed to make them work. Perceived complexity derives from the particular application and the kind of information involved, not from the elements of user interaction in *Pan*'s design framework through which they can be delivered.

5.8.1 Example Applications

This section describes *Pan* applications that have been prototyped in some way. A few have been mentioned elsewhere, but are included here for completeness.

Structural Search/Replace Like all powerful text editors, *Pan* supports textual replacement based on regular expression matching. However, one sometimes wants replacements to depend on the language structure, not on the textual structure, even when the two are similar. For example, whole-word replacement (where replacing substrings of longer words is not desired) in natural language documents is difficult to specify using patterns. One variant of *Pan*'s replacement command matches patterns only against words (lexemes) as defined by the particular language. Another variant renames variable instances in programs, drawing on information in *Pan*'s database to avoid renaming enclosed variable definitions that have the same lexical name but which are logically different variables.

Queries and Non-Local Navigation One of the few forms of query supported by ordinary text editors is textual search. Searching in *Pan* can draw on *any* derived information. For example, one command locates the declaration and all uses of a program variable, which the user identifies by pointing at any instance. The results of this and other structure-based queries are made available using the query interface described in Section 4.3.8, including a highlighter and operand level for navigation over the results. This particular command is implemented by using a database query defined as part of the *Colander* description in the view style specification. The same underlying *Colander* query supports other services, for example a command that moves the cursor to the declaration of a variable pointed to by the user; this is only one example of a navigation command that follows hypertext-like links defined by the underlying language.

Names, Types, and Values Other sample applications exploit information that is derived in the process of checking a language's static semantics. The cross reference and table of contents view styles (Section 4.3.11) are among these. Another command announces (in the panel) the type of a name, and another announces its value if known. A variant of the latter displays the value temporarily in place of the name. Value computations could be augmented by other sources of information and more advanced reasoning. For example, expressions might be evaluated when enough values are known, and the user might direct that certain assumptions about variable values be made (these might be context- or name-specific "rules," for example to assume that a particular error flag is never true).

Style Checking Stylistic and usage constraints, added to the *Colander* description in a language-based view style, specify violations that do not show up as language errors. Violations can can be viewed by adding a standard panel flag, a standard highlighter, and by navigation at level "`Stylistic Violation`".

Textual Annotations The user can create textual annotations, in a special alternate view for each. A separate summary view lists all existing annotations, and supports shared structural navigation to help the user relate annotation and document component (Section 4.3.11). A programmatic client interface for the facility exists, which might permit a debugger, for example, to annotate various components with context information derived from a stack trace being studied.

Debugger Integration An experimental debugger permits code modifications to be compiled incrementally from *Pan*'s internal representation and patched into an executing image while preserving execution state [18].

5.8.2 Easy Additional Applications

This section describes applications that would not be difficult to implement within *Pan*'s existing framework.

Classes for Library Calls An operand class might be defined to include call sites involving particular libraries, for example calls to window system libraries. Basic services would let users highlight call sites, navigate over them, or generate a separate list view showing all calls. A specialized command available for this operand class might display documentation for a selected library function.

Classes for Particular Language Constructs As part of a software re-engineering task, operand classes might be defined for particular constructs of interest, for example all uses of "goto," allowing rapid determination of their presence, their number, their location, or perhaps the diversity of their target destinations.

Classes for Other Annotations Operand classes might be defined for any other class of information that could be imported from tools in the environment, for example dead code analysis, test coverage, and many others.

Type Checking for Complex Argument Sequences The conventional static-semantic goals for a language, for example ordinary type checking in C, could be extended to check complex, dynamically defined argument sequences to procedures, for example the attribute/value list discipline used by the XView libraries [53]. This argument discipline calls for alternating keywords (defined constants) and values, where the type of the value must be appropriate for the attribute. Some values are sublists, which must be terminated by an argument of zero, and some attributes take no values. None of this can be checked by standard type systems, but they could be encoded into a set of *Colander* constraints specialized for the library.

Current Modifications An operand class might be defined to include all new structure created since some previous reference point (originally visiting the file, or perhaps a user specified reference point). Either a highlighter or a query would make it possible to see code newly introduced during the current session.

5.8.3 Complex Additional Applications

This section describes more speculative *Pan* applications. These are the kind of applications that would exploit more fully *Pan*'s basic design framework, but all would require additional mechanisms and possibly integration with outside tools.

Graphical Cues in Display Small glyphs or icons might be associated with particular flavors (or layers) of structural annotations or properties, and these might appear in the display at each site where one is present. Baecker and Marcus suggest something similar for static document display in their "Essay on Comments," complete with twenty suggestive examples for categories such as "Warning/Sensitive," "Added Code," "Fragile Code," and "Unreachable Code" [6]. This application would require a more flexible presentation engine than *Pan*'s current prototype.

Extra Notation in Display Redundant notation generated by the system can aid comprehension. For example, certain scope markings have been shown helpful for some kinds of reasoning [117] but not for others. Studies such as these focus on language design, in this case examining the tradeoff between redundancy (and therefore more opportunity for entry errors) and comprehension. Generating the extra notation automatically (and optionally) in a system like *Pan* gives the user the best of both choices. This application would require a more flexible presentation engine than *Pan*'s current prototype.

Transformed Notation in Display Notational transformations, produced by the system, can aid comprehension, especially when placed *in situ* in the display so that continuity with surrounding context is not disturbed. These would be helpful for constructs known to be confusing, for example deeply nested conditionals which can be transformed into decision tables or other representations advocated for manual use [99]. This application would require a more flexible presentation engine than *Pan*'s current prototype, as well as some special-purpose analyzers.

Slicing Program slicing is a form of visual program condensation based on the question "what can affect the value of this variable at this point in the program" [137]. A family of queries of this sort might be answered by a general analyzer. This application would require a more flexible presentation engine than *Pan*'s current prototype, as well as integration with a data-flow analyzer.

Control Folding in Display A specialized transformation would elide certain control branches, producing a "collapsed view" based on contextual assumptions. Most helpful would be the assumption that no error-handling branches are taken, yielding a normal-case view to help explain the main program. Identification of error-handling branches might take place by inference, based on pattern matching and stylistic rules, by importation of profile information, by direct annotation from users, or by a mix of these. This application would require a more flexible presentation engine than *Pan*'s current prototype, as well as possible integration with a profiler.

General Debugger Interface *Pan* would serve well as the source code viewing component of a generalized debugger (perhaps multi-language, as is the rest of *Pan*). Special glyphs, mentioned earlier, would mark control and break points in the customary fashion, but *Pan*'s other services would also be available for the user to browse and query. The debugger could be requested to store a stack trace by automatically generated annotations, for example stack frame information at relevant nodes.

Specialized Debugger Interface *Pan* is flexible enough to serve as an interface to many specialized tools, for example a recently developed debugger for optimized code [19]. The debugger displays both the source (the primary view in *Pan*) and the optimized assembly code (an alternate view in *Pan*), with jointly visible stepping (shared navigation in *Pan*). In this debugger certain steps manifest themselves as discontinuous components in the source code because of compiler optimization, a prospect that would be handled by the kind of multi-structural selection described in Section 5.3.4.

Improved Diagnostics A high level diagnostic service, tunable with declarations based on experience, would use as much information as possible (including perhaps judiciously stored history information in the database) to produce better explanations of language errors than is now possible.

Chapter 6

Experience

This chapter reviews some aspects of *Pan*'s design framework in which research results have been mixed and where useful lessons can be learned. The first such aspect is the relationship between *Pan*'s enabling technology (imported and adapted from compiler technology) and the demands of user-centered design. Although considerable progress was made in this area, the legacy of the batch-oriented compiler still causes problems in *Pan*. A second aspect is the difficulty that comes with designing and building a completely new text editor. The cost was extremely high, the result still isn't good enough, and yet no other known editor (neither when the project began nor now) would meet *Pan*'s needs. The third aspect includes a number of observed shortcomings of *Pan*'s mechanisms and user elements for language-based interaction.

6.1 Porting Compiler Technology

From this effort to apply user-centered design have emerged insights concerning the demands that it places on the enabling technology, and in particular on the *Ladle* and *Colander* subsystems in *Pan*. This adds to general evidence, exemplified by Shaw's discussion of input/output mechanisms [115], that underlying computing models designed for batch execution are characteristically unsuited for interactive systems. This section articulates these special demands, explains how advances in the technology (including *Ladle* and *Colander*) have met them partially, and discusses ways in which they remain unsatisfied.

A recurring theme is the failure of the analyzers to capture and exploit the complete *context* of user interaction. For example, incremental algorithms are typically designed to bound the context (and thereby the amount) of computation necessary to process each increment of change by the user [141]. In practice users' changes are related to other user actions separated by space and time. Failure to capture the broader context of interaction causes *loss of information* in surprising (and annoying) ways. Consider for example nesting delimiters **BEGIN** and **END**, which might in an incremental system become unmatched either through addition of a **BEGIN** or deletion of an **END**. Most parsers would produce the same diagnostic in either case, for example "**Unmatched BEGIN**", but to the user who has just accidentally deleted an **END** this message suggests that the system ignores the obvious.[1]

[1]This example is from Wegman and Alberga [136], whose incremental parser would produce different messages in the two cases, for example "**Missing END**" in the latter.

This is a small example, and it has little consequence for users experienced at deciphering messages encoded in the compiler's frame of reference, but there are many more examples like it; each adds a small amount of unnecessary cognitive overhead to the user's task. As with this example, none of the issues raised in this section translate directly into visible breakage in (or absence of) particular services. Rather, each is a potential degradation in overall capacity of the system to provide effective services.

The fact that the issues raised here are *not* directly visible to users is an advantage of the boundary between the lowest two design layers in *Pan*'s implementation model (Figure 3.7 on page 30). This observation renders the issues no less problematical, but it does help characterize a large class of interaction problems. The rest of this section addresses specific challenges to *Pan*'s language-based infrastructure, organized for exposition and not necessarily in order of difficulty or consequence.

6.1.1 Convenient Language Description

Following *Pan*'s requirement for "description-driven support for multiple languages" both language-based analyzers are driven by declarative descriptions, and any number of descriptions may be loaded into a running system (Section 4.1). Writing LALR grammars isn't simple, but *Ladle* eases the burden by permitting primary specification in terms of an abstract syntax (with control over details of the tree representation to be maintained); with only a few additional rules *Ladle*'s preprocessor is able to infer a complete parsing grammar. The *Colander* description language likewise achieves perspicacity with a language modeled on goal satisfaction and named collections of facts [9]; an explicit design goal for that model was to enable descriptions that read naturally, more so for example than attribute grammar specifications [28,105] or action routines [64,84].

Since these goals are common to both batch compilation systems and *Pan*, advances such as these represent mutual progress. Even so, convenient language description remains an elusive goal. For example, *Pan*'s distinction between syntax and contextual constraints (static semantics) is common to almost all language description techniques because of the simplicity it achieves for both description and implementation. But it creates problems in practice for badly designed languages in which, for example, parsing and semantic analysis must be intertwined [35,85], as with the well known "typedef" problem in the C language. Furthermore, experience with *Colander* descriptions suggests that the language, while appropriately expressive, embodies a level of abstraction too low for convenience. Research into general solutions to these problems within *Pan* is currently underway.

Meanwhile, problems that arise in the interactive world confound the picture. For example, there is a need for *description sharing*, both piecewise and in whole, that is poorly addressed by current technology, *Ladle* and *Colander* included. This need arises most simply in *Pan* when two or more different view styles, perhaps written for different users or tasks, are based on identical underlying language descriptions. Slightly more complex examples include the design of view styles with separate concrete grammars but a shared abstract syntax, and the design of view styles with various partial descriptions for static-semantic checking. More complex yet would be the capture of general syntactic and static-semantic abstractions for reuse when describing related languages. Finally, there is a need for descriptions that cross traditional language boundaries, linking together parts of software systems that are realized in more than one language (including documentation languages).

6.1.2 Unconstrainted Text Editing

Pan's requirement for "familiar, unrestricted text editing" demands an unrestricted "smart text editor" that also supports language-based interaction. *Pan* is **syntax-recognizing**, as are Babel [57], the Saga editor [67], and SRE [21]: the user provides text and the system infers the syntactic structure by analysis. An implementation problem with this approach has been the necessity to maintain large and unwieldy syntactic representations for incremental reanalysis. The theory of *grammatical abstraction* [10,22] was developed to resolve this problem in *Pan* (Section 4.1.2).

In contrast, **syntax-directed** systems such the Cornell Program Synthesizer[125], Mentor [29], and Gandalf [50] require that the user explicitly construct internal structure by selecting syntactic templates. The historical distinction between the two approaches reflects divergent goals: users prefer no restrictions on text-based editing, but implementors prefer that modifications be expressed in terms of atomic changes to internal representations.

To users the distinction between the two styles is becoming less sharp. Syntax-directed systems are evolving toward hybrids with increasing text-based support, beginning with Emily [52], which supported almost none, through recent incarnations of the Synthesizer Generator [103], which are increasingly flexible about the context (still structurally defined) in which text editing may take place. Conversely, *Pan* can support a syntax-directed style of editing superimposed on (and coexisting with) the syntax-recognizing model, as described in Section 5.5.

To implementors, however, the distinction continues to reflect design tradeoffs. These don't exist in the world of compilers, where analysis is usually a batch-oriented operation in which history plays no role. The tradeoffs become acute in the presence of incrementality and error recovery, issues to be discussed subsequently.

6.1.3 Incremental Analysis

Incremental computation is conventionally held to be important for interactive program development environments, since useful information must be derived from constantly changing programs. The guiding principle has been that performance gains can be realized by minimizing the amount of computation performed in response to each change [141]. Naive application of this principle to systems like *Pan*, however, ignores the full context of interaction and produces system behavior that is unacceptable to users.

The problem arises from modeling such systems as interactive compilers whose input (program text) is constantly changing and whose behavior depends *only* on the input. This model is inadequate for *Pan*, where users and other applications create ephemeral but useful information superimposed on the linguistic structure of a program. One can think of this information as the state of an ongoing *conversation* between the user and the system. As in any conversation it is important that the referents of the conversation be stable and persistent; incremental analyzers must maintain the *identity* (as opposed to only the equivalence) of program components in the presence of change, a goal not addressed by standard implementations. Systems that support conversations among people encounter similar problems, for example users' "difficulty determining that they were talking about the same objects..." [124].

Ladle's incremental parser, for example, is based on an algorithm that first "unzips" the tree along a path between the leftmost change site and the root; when the algorithm eventually reconstitutes the tree, using new nodes as needed, it has effectively discarded the identity and any annotations associated with nodes along that path. This has serious consequences even from the strictly compiler-oriented point of view. Widely-shared contextual data often appears closer to the root of the tree, and its loss would incur lengthy and often unnecessary recomputation. *Ladle*'s parser ameliorates this effect with a heuristic for reconstituting unzipped nodes: "divided" tree nodes are kept on a special stack and may be reused whenever a node represents the same production in the abstract syntax as in its previous use and when its leftmost child is unchanged between parses. Although effective at restoring tree nodes near the root, this heuristic doesn't always maintain identity in the vicinity of small changes.

Ladle's failure to retain nearby context derives from a deeper problem related to the origin of the technology. This incremental parser is designed as a batch-oriented left-to-right parser with the added ability to be *restarted* at any intermediate point. In the vicinity of a change, the parser examines the new text stream but ignores the *difference* between old and new, which is the real context of the user's modification. This weakness leads to the "`Unmatched BEGIN`" example cited at the beginning of this section.

Earlier generations of syntax-directed editors avoided the problem by permitting textual editing only in the narrowest of contexts, expressions or statements for example [66,126]. But the hybrid approach, which syntax-directed editors have had to adopt for reasons of usability, encounters a variation on the same problem. All textual changes take place in a structurally defined "focus" that amounts to "destructured text"; at reanalysis time it is precisely the local structure that has been discarded and must be reconstrued by the parser.

Because of this weakness, no *Pan* services that depend on identity are robust in the immediate neighborhood of changes, however small. This is a partial failure to meet *Pan*'s requirement for "uninterrupted service".

6.1.4 Granularity of Analysis

In a syntax-recognizing system, the choice of policy for initiating reanalysis has profound implications for the underlying technology. *Pan*'s policy is lazy, only analyzing when the information is needed or when the user requests it explicitly. This implies that derived data and the text stream are often *inconsistent*, which in turn creates problems for user interaction that are described in Section 5.6. It also leads to situations (not anticipated in the design of either *Ladle* or *Colander*) where derived data must be made available, even though it is incomplete and unreliable, during periods of inconsistency.

An opposite extreme, adopted for the Magpie editor [111], is to run the analyzer after every keystroke. This system needs no machinery to deal with inconsistency, but the analyzers must be very fast and extremely adept at dealing with malformed programs (since at this granularity programs are nearly always malformed). Magpie's analyzer is fast, but it drastically curtails analysis upon encountering a single language error and provides fewer services in general. Furthermore this approach exacerbates the problems faced by *Pan*'s analyzers in tracking the identity of program components during change.

The compromise offered by hybrid syntax-directed editors such as the Synthesizer Generator[103] and Gandalf [50] permits textual change only within a structurally defined focus. This bounds the structural context of text-based changes, but typically prohibits navigation away from the focus until its text has been parsed successfully without appearance of language errors; this is an unacceptable interruption of service.

6.1.5 Tolerance for Language Errors

Pan's lazy analysis policy ensures that analyzers are confronted with malformed programs less often than would be the case following other policies. Users tend to request analysis or invoke language-based commands only when they judge their recent changes to be at least locally error-free. Even so, language errors are common and *Pan*'s requirement for "uninterrupted service" applies in their presence. One crucial service in the presence of language errors is the presentation of diagnostics, discussed in Section 5.4. For this and other services to continue operating at all in the presence of errors places pervasive demands on the language-based infrastructure.

Pan's design implicitly extends internal representations to subsume ill-formed programs, and a design goal for *Ladle* and *Colander* was to treat them as interesting but relatively normal occurrences. This happens uniformly, without imposing excessively on language description authors, and it interferes little with other language-based interaction.

Whenever lexical analysis fails, a lexical variance is signaled. For instance, an unterminated comment may lead to a lexical variance. A variance detected during lexical analysis inhibits both parsing and contextual-constraint checking, and all information that existed prior to the attempt to reanalyze the document is preserved.

During parsing *Pan* uses a panic mode mechanism [26] for syntactic error recovery. Directives in *Ladle* descriptions tune the recovery mechanism for each language. The presence of a syntactic variance is marked in the internal tree by an **error subtree** annotated with an appropriate error message. The children of an error node are the lexemes and subtrees that were skipped over during the recovery. This recovery strategy is similar to that used in the Saga editor [67]. Any extant annotations on the subtrees within the error subtree are preserved, including annotations created by *Colander*. Contextual constraints within an error subtree are not attempted by *Colander*, but remain available for possible reuse after correction of the variance. Unsatisfied contextual constraints detected by *Colander* are also variances, resulting in annotations on offending nodes.

Experience suggests that these mechanisms are still inadequate. Panic-based recovery inherits a deficiency, mentioned earlier, of the simple left-right parsing algorithm: it only considers the new text and not the actual change. A more thorough analysis of the changes would permit better heuristics for narrowing the scope of error nodes. Even more problematic, however, the system is unable to restore information (and in particular the identity and annotations of nearby nodes) lost during the presence of even ephemeral and trivial errors. For example, a carelessly typed character adjacent to the outermost `BEGIN` in a program would cause the analyzers to discard nearly all of the useful information about a program; even after an immediate correction (prompted by diagnostic results) expensive and possibly irreplaceable information might be lost. This information loss is an interruption of service, caused by another failure to capture the entire context of user interaction (in this case closely related changes separated by time).

Finally, *Ladle*'s implementation reflects a tendency in the parsing literature to treat error recovery and diagnosis together (to the detriment of both in this case). In *Pan*'s layered implementation model, however, they should be assigned to different layers reflecting different demands being placed upon them. The goal of error *recovery* (part of *Pan*'s infrastructure) is to produce a representation of a badly-formed program with as little loss of information as possible. The goal of error *diagnosis* (a basic user service in the third layer) is to help the user locate, understand, and possibly correct the problem. The two mechanisms are strongly related, both must exploit contextual information, both could be at least partially heuristic, and both must be tuned (perhaps per-language) based on experience. There is no reason, however, to expect that the same mechanisms and configurations will address both goals completely.

6.1.6 Broad Domain of Analysis

Colander's analyzer was designed to be general purpose as well as multi-lingual [9]. Its expressive power suffices for static-semantic checking in programming languages, but it can and has been used for other purposes. For example, an experimental constraint on variable naming conventions required only the addition of a new built-in operator (a regular expression pattern matcher) to the Colander language; since a pattern matcher was already available in the text editor, it was easily added to *Colander* using its internal language extension facilities. But experiments also suggest that finer-grained control over the *Colander* analyzer will be needed as more kinds of analysis are added. One might wish to check language-based constraints frequently while deferring more complex tests. *Colander* analysis at present is monolithic, however; all potentially unsatisfied constraints must be attempted at each invocation.

Experience with *Colander* has revealed other limitations:

- A data flow analyzer implemented in Colander required a convoluted description and resulted in inefficient computation.
- An investigation of stylistic guidelines revealed that non-local structural relationships such as "before" and "after" do not find natural expression in *Colander*, and that some might better be described as tree patterns than as *Colander* goals.
- As described in Section 4.2.6, the *Colander* specification language has no provision for requesting incremental computation of operand class membership and for propagating changes in class membership. General support should be added to the analyzer at a low level.
- Diagnostics are failure-based, so that information *about* unsatisfied constraints is handled specially and is not available uniformly in the database.
- There is no first-class way to deal with partial knowledge in *Colander*. There are no tools to help the author control propagation of failures and spurious diagnostics, so this must be done manually. Similar support for placeholders (Section 5.5) would be helpful.

Whether these would be best addressed by extensions to *Colander* or by additional analyzers (for example, a tree pattern matcher like the one used for prettyprinting in the Mentor system [63,89]) is open pending further research.

6.1.7 Data Sharing

One reason additional analyzers can be considered at all is that *Colander*'s database was designed to be shared with other tools. For example, a prototype context-based prettyprinter attached information to nodes and relied on *Colander* for incremental data propagation [16]. A relatively open data model is crucial for the kinds of user-oriented services that fully exploit the potential for such systems.

Not enough of this sharing was done in *Pan*, however. For historical reasons, *Ladle* and *Colander* use separate data representations, even though much of the structural information ends up being replicated. Aside from the obvious inefficiencies, clients suffer from lack of a uniform access mechanism.

6.2 Building a Text Editor

One of the costliest and perhaps most frustrating parts of the *Pan* project was the creation of an entirely new text editor. It takes a tremendous amount of effort to build and maintain an editor whose textual facilities even approach those of GNU EMACS, which is available without cost. At the time the project began, however, no editor available had the facilities needed, for example an open architecture, a rich internal text representation, and a display engine capable of rendering multiple proportionally spaced fonts using colorful highlighting effects. Section 5.1 discuss *Pan*'s achievements in the effective use of fonts and colors. More recently developed GNU EMACS variants, for example Epoch, have added some facilities of this kind and may be more promising platforms for a system like *Pan*.

6.2.1 Needed Improvements

At the same time, however, *Pan*'s rendering engine remains inadequate; it has constrained the services that could be built as part of this research. All of the following would also be useful:

- Selective elision, including holophrasting [3,119];
- Direct spatial layout, free from the vagaries of explicitly represented whitespace characters (spaces, tabs, and newlines);
- Spatial modes, for example wide margins in which any inserted text is presumed to constitute a comment;
- Graphical cues, including rules (lines), boxes, long arrows, and shading, all used to suggest relationships among textual components [6];
- Special non-textual symbols to annotate text, for example pointing hands, and traffic signs [6]; and
- Nested displays, for example, to permit a program component to be displayed *in situ* using a different (perhaps graphical or tabular) formatting discipline.

Many of these are being addressed as part of the Ensemble project, *Pan*'s successor, by combining high-quality document presentation with *Pan*'s language-based services [48].

6.2.2 Toolkit and UIMS Support

Window system toolkit support for *Pan* was never especially helpful for the most difficult part of the window interface: the text rendering engine. Successive versions of *Pan* were implemented upon Suntools, SunView, and the Athena Widget Set. At each stage text rendering was implemented using raster-level primitives, since nothing better was (or is) available.

More attention is being directed to the development of user interface management systems (UIMS), which permit mappings between internal data, behaviors, and external appearance using declarative specifications [55,58]. It seems unlikely that one would ever support *Pan*'s richly overloaded text display, since it acts as the output channel for so many different kinds of information simultaneously.

6.3 Other Design Issues

Other problems and lessons have become apparent for each of *Pan*'s four design layers, as presented in Figure 3.7 on page 30. The first section in this chapter concerned the infrastructure layer. This section addresses the remaining three.

6.3.1 Kernel

The most glaring omission from *Pan*'s kernel services is the absence of a general mechanism for propagating changes. For historical reasons *Ladle*'s parser, *Colander*'s analyzer, the text representation and rendering engine, scoped editor variables, operand class membership computation, and highlighting all evolved separate mechanisms. Although the interfaces among these components have been carefully controlled as the system evolved, a more general paradigm for event propagation would make their implementations more tractable. One unexplored possibility would be to expand the role of *Colander*'s database and inference engine, to provide low-level support for these functions in a more general way.

6.3.2 Basic Services

Experience has revealed weaknesses in some of the elements of user interaction described in Section 4.3. For example, the structure cursor has been the subject of ongoing development, although the version described in Section 4.3.7 represents a stable point in its evolution. Selection, on the other hand, has proven to be a design problem severe enough to hinder development of some otherwise implementable prototypes. The next evolutionary stage for this aspect of *Pan*'s design, described in Section 5.3.4, would be a mechanism for multiple-component textual selection, with an optional structural aspect in the same sense as the cursor.

Pan's panel flag mechanism has proved useful and robust throughout most applications, but there have been cases where more than two flag states would be appropriate. The flag service was always based on simple boolean operation, so the extension would involve changes at every level from implementation through specification.

6.3.3 View Styles

A number of interesting problems become apparent at the view style layer, where the interface between a class of users and a class of documents must be specified and fine-tuned.

For example, the choice of colors to satisfy various design guidelines has a large impact on the success of *Pan*'s design metaphors, but the decisions can involve subtle differences of colors to get them just right. Unfortunately, color perception is subject to variations in display screens, ambient light, and individual differences, factors that can dominate the subtle color distinctions that are most effective. Convenient personal adjustment is therefore important, but window system support for color specification is crude,[2] and where the *relationships* among colors are paramount, that support is nonexistent. More helpful would be some way for basic color schemes (*sets* of colors in a *Pan* color map, for example) to be continuously adjustable by a single control; thus, a user might dynamically turn the colors up or down on the display, adjusting for local conditions without affecting the basic color assignment scheme. A second control might adjust relative hue selection, much like Tognazzini's "Relative Color" proposal [129].

Font choices are also difficult; as discussed in Section 5.1, none are really suitable. Bigger fonts are much nicer, but take up too much space. Punctuation is especially hard to read in some languages, for example *Colander*, where it is very important to distinguish between commas and periods. Punctuation characters in proportional spaced fonts tuned for text aren't very satisfactory in general for programming languages.

A final meta-problem concerns the fundamental premise of view style design in *Pan*. As the many guidelines in Section 4.4.3 suggest, and the discussions in Chapter 5 confirm, careful configuration is essential to successful implementation of *Pan*'s design metaphors. In *Pan*'s design framework this is considered to be a design task, not necessarily to be carried out by every user. At the same time, however, the requirement for "extensibility and customizability" dictates that the user be permitted to customize freely. Thus any user might, mindfully or not, degrade the effectiveness of *Pan*'s fundamental services with careless or ill-advised customizations. The only practical solution (short of the unacceptable restriction of user rights) rests with plentiful alternative view styles for users to choose among and with presumed close collaboration between the kind of local experts described in Section 4.4.1 and their user population.

[2] *Pan*'s color maps are now specified by selection from a standard X window system list of idiosyncratically named colors.

Chapter 7

Conclusion

This dissertation began with the hypothesis that language-based editing, when implemented with suitable attention to user-centered design principles, can become a practical, usable component in the working environment of experienced software professionals. Investigation of this hypothesis began with a reconsideration of the technology and the assumptions, both stated and unstated, about how it should best be used. From that reconsideration emerged a proposed set of design requirements for such a system.

These requirements have complex implications that often conflict; worse, they threaten to drown users in a sea of system-induced complexity when exactly the opposite effect is needed. 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, permits such a system to be built.

The *Pan I* Version 4.0 prototype described in this dissertation meets the proposed requirements:

1. *Pan* is a practical, powerful, text editing system whose conventional functionality is never restricted by the addition of language-based features.

2. *Pan* presents documents in a rich text-based information display using multiple, proportionally spaced fonts with potentially many kinds of metainformation (information *about* documents) superimposed via colored highlighting.

3. *Pan*'s interface between user and system is built around a small number of easily understood and flexible services whose apparent complexity derives only from the kind of information being displayed and the task at hand; view styles in *Pan* specify interfaces between users and documents that are tuned for particular tasks and languages.

4. *Pan* supports multiple simultaneous views of software documents, presenting different kinds of information organized in different ways; the framework permits creation of more such views as additional kinds of information become available.

5. *Pan*'s services degrade gracefully when documents contain language-errors, when they are incomplete, and when analysis-derived information is out of date.

6. *Pan* can load any number of declarative language-based view style definitions on demand, each including a *Ladle* description, a *Colander* description, and a specification for user interaction.

7. *Pan* is customizable, extensible, and programmable; it has proven to be an effective, flexible research platform.

The *Pan I* prototype continues to support ongoing research at UC 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 [48].

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

Isolation of Language-Based Technology It is tempting think of language-based editing systems as interactive compilers, but the needs of programmers are dramatically different from those of compilers. Language-based technology developed for compilers ports badly into the domain of user interaction, 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. *Pan*'s user-oriented services work equally well in an alternate version of the system built with an experimental replacement for the *Colander* analyzer [18].

Operand Class Abstraction The operand class has proven to be an effective abstraction. The description-driven operand class 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. Examples include highlighters, panel flags, projection views, and the dual-aspect cursor.

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 its users, the medium in which they work, and 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.

This research suggests further investigation in several areas.

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 This research suggested many potentially useful presentation techniques based on the static book publishing paradigm. Many of those techniques are not supported by *Pan I*'s prototype rendering engine, but are being explored by the Ensemble project. 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 research 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 interconnectivity.

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 COMMON LISP. 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 research. 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.

References

[1] James Ambras & Vicky O'Day, "MicroScope: A Knowledge-Based Programming Environment," *IEEE Software* 5 (May 1988), 50–58.

[2] Apple Computers, Inc., *MacWrite Manual*, Cupertino, California, 1984.

[3] J. E. Archer, Jr. & R. Conway, "Display Condensation of Program Text," *IEEE Transactions on Software Engineering* SE-8 (September 1982).

[4] James Archer, Jr. & Richard Conway, "COPE: A Cooperative Programming Environment," Department of Computer Science, Cornell, Technical Report 81-459, June 1981.

[5] Yigal Arens, Lawrence Miller, Stuart C. Shapiro & Norman K. Sondheimer, "Automatic Construction of User-Interface Displays," *Proceedings Seventh National Conference on Artificial Intelligence* (August 1988).

[6] Ronald M. Baecker & Aaron Marcus, *Human Factors and Typography for More Readable Programs*, ACM Press, 1990.

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

[8] Rolf Bahlke & Gregor Snelting, "Design and Structure of a Semantics-Based Programming Environment," *International Journal of Man-Machine Studies* 37 (October 1992), 467–479.

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

[10] Robert A. Ballance, J. Butcher & 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 (June 22-24, 1988), 185–198.

[11] Robert A. Ballance & Susan L. Graham, "Incremental Consistency Maintenance for Interactive Applications," in *Proceedings of the Eighth International Conference on Logic Programming*, K. Furukawa, ed., MIT Press, Cambridge, Massachusetts, 1991, 895–909.

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

[13] Robert A. Ballance, Michael L. Van De Vanter & Susan L. Graham, "The Architecture of Pan I," Computer Science Division, EECS, University of California, Berkeley, 88/409, August 1987.

[14] Jon Bentley, "Little Languages," *Communications of the ACM* 29 (August 1986), 711–721.

[15] Ted J. Biggerstaff, "Design Recovery for Maintenence and Reuse," *Computer* 22 (July 1989), 36–49.

[16] Christina L. Black, "PPP: The Pan Program Presenter," Computer Science Division, EECS, University of California, Berkeley, 90/589, November 1990, Master's Thesis.

[17] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang & V. Pascual, "CENTAUR: the system," *Proceedings ACM SIGSOFT '88: Third Symposium on Software Development Environments* (November 1988).

[18] John Boyland, Charles Farnum & Susan L. Graham, "Attributed Transformational Code Generation for Dynamic Compilers," in *Code Generation – Concepts, Tools, Techniques*, Robert Giegerich & Susan L. Graham, eds., Workshops in Computing Series, Springer Verlag, Berlin, 1992.

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

[20] Ruven Brooks, "Using A Behavioral Theory of Program Comprehension in Software Engineering," *Proceedings 3rd International Conference on Software Engineering* (May 1978).

[21] Frank J. Budinsky, Richard C. Holt & Safwat G. Zaky, "SRE – A Syntax Recognizing Editor," *Software–Practice & Experience* 15 (May 1985), 489–497.

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

[23] John M. Carroll & John C. Thomas, "Metaphor and the Cognitive Representation of Computing Systems," *IEEE Transactions on Systems, Man, and Information Sciences* SMC-12 (1982), 107–116.

[24] Ravinder Chandhok, Phillip Miller, John Pane & GlennMeter, "Structure Editing: Evolution Towards Appropriate Use," *Presented at the CHI '90 Workshop on Structure Editors*, Pittsburgh, Pennsylvania (April 1990), Position Paper.

[25] Jacques Cohen, "Constraint Logic Programming Languages," *Communications of the ACM* 33 (July 1990), 52–68.

[26] Robert Paul Corbett, "Static Semantics and Compiler Error Recovery," PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, 85/251, June 1985.

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

[28] Pierre Deransart, Martin Jourdan & Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, Lecture Notes in Computer Science #323, Springer Verlag, Berlin, August 1988.

[29] Veronique Donzeau-Gouge, Gerard Huet, Giles Kahn & Bernard Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe & Erik Sandewall, eds., McGraw-Hill, New York, NY, 1984, 128–140, Chapter 7.

[30] Sarah A. Douglas & Thomas P. Moran, "Learning Text Editor Semantics By Analogy," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts (December 1983).

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

[32] J. Doyle, "A Truth Maintenance System," in *Readings in Artificial Intelligence*, Bonnie Lynn Webber & Nils J. Nilsson, eds., Tioga, Palo Alto, California, 1981, 496–516.

[33] H. John Durrett, ed., *Color and the Computer*, Academic Press, New York, NY, 1987.

[34] Michael J. Fischer & Richard E. Ladner, "Data Structures for Efficient Implementation of Sticky Pointers in Text Editors," Department of Computer Science, University of Washington, 79-06-08, June 1979.

[35] Bruce T. Forstall, "Programming Language Specification for Editors," Computer Science Division, EECS, University of California, Berkeley, Master's Thesis, November 1991.

[36] Christopher W. Fraser, "Syntax Directed Editing of General Data Structures," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 17–21.

[37] Christopher W. Fraser, "A Generalized Text Editor," *Communications of the ACM* 23 (March 1980), 154–158.

[38] Christopher W. Fraser & A. A. Lopez, "Editing Data Structures," *ACM Transactions on Programming Languages and Systems* 3 (April 1981), 115–125.

[39] James A. Galambos, Eloise S. Wikler, John B. Black & Marc M. Sebrechts, "How You Tell Your Computer What You Mean: Ostension in Interactive Systems," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts (December 1983).

[40] E. H. Gansner, J. R. Horgan, D. J. Moore, P. J. Surko, D. E. Swartwout & J. H. Reppy, "SYNED – A Language-Based Editor for an Interactive Programming Environment," *IEEE Computer Society 7th International Computer Software and Applications Conference* (1983).

[41] Michelle Gantt & Bonnie A. Nardi, "Gardeners and Gurus: Patterns of Cooperation Among CAD Users," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Monterey, California (May 3-7, 1992).

[42] David Garlan, "Flexible Unparsing in a Structure Editing Environment," Computer Science Department, Carnegie-Mellon University, CMU-CS-85-129, Pittsburgh, Pennsylvania, April 1985.

[43] David Garlan, "Views for Tools in Integrated Environments," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-87-147, Pittsburgh, Pennsylvania, May 1987.

[44] Brian Gill-Price, "Personal Communication," 1992.

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

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

[47] John D. Gould, Lizette Alfaro, Rich Finn, Brian Haupt, Angela Minuto & Josiane Salaun, "Why Reading was Slower from CRT Displays than from Paper," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Toronto, Canada (April 1987).

[48] S.L. Graham, M.A. Harrison & E.V. Munson, "The Proteus Presentation System," *Proceedings ACM SIGSOFT '92: Fifth Symposium on Software Development Environments* (December 1992).

[49] Jonathan Grudin, "The Case Against User Interface Consistency," *Communications of the ACM* 32 (October 1989), 1164–1173.

[50] A. Nico Habermann & David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering* SE-12 (December 1986), 1117–1127.

[51] Heikki Halme & Juha Heinanen, "GNU Emacs as a Dynamically Extensible Programming Environment," *Software–Practice & Experience* 18 (October 1988), 999–1009.

[52] Wilfred J. Hansen, "User Engineering Principles for Interactive Systems," in *Interactive Programming Environments*, David R. Barstow, Howard E. Shrobe & Erik Sandewall, eds., McGraw-Hill, New York, NY, 1984, 217–231, Chapter 11.

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

[54] Paul N. Hilfinger & Phillip Colella, "FIDIL: A Language for Scientific Programming," in *Symbolic Computation: Applications to Scientific Computing*, Robert Grossman, ed., SIAM, 1989, 97–138.

[55] Deborah Hix, "Generations of User-Interface Management Systems," *IEEE Software* 7 (September 1990), 77.

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

[57] M. R. Horton, *Design of a Multi-Language Editor with Static Error Detection Capabilities*, PhD Dissertation, Electronics Research Laboratory, EECS, University of California, Berkeley, 1981.

[58] Robert J. K. Jacob, "A Specification Language for Direct-Manipulation User Interfaces," *ACM Transactions on Graphics* 5 (1986), 283–317.

[59] Fahimeh Jalili & Jean H. Gallier, "Building Friendly Parsers," *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico (January 1982).

[60] Vera John-Steiner, *Notebooks of the Mind: Explorations of Thinking*, Harper & Row, 1985.

[61] Jeff Johnson, "Modes in non-computer devices," *International Journal of Man-Machine Studies* 32 (April 1990), 423–438.

[62] Jeff Johnson & George Engelbeck, "Modes Survey Results," *ACM SIGCHI Bulletin* 20 (April 1989), 38–50.

[63] Giles Kahn, Bernard Lang, Bertrand Melese & Elham Morcos, "Metal: A Formalism to Specify Formalisms," *Science of Computer Programming* 3 (1983), 151–188.

[64] Gail E. Kaiser, "Semantics of Structured Editing Environments," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-85-131, Pittsburgh, Pennsylvania, 17 May 1985.

[65] Gail E. Kaiser, Peter Feiler, Fahimeh Jalili & Johann H. Schlichter, "A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation," *Software–Practice & Experience* 18 (August 1988), 733–748.

[66] Gail E. Kaiser & Elaine Kant, "Incremental Parsing without a Parser," *Journal of Systems and Software* 5 (May 1985), 121–144.

[67] Peter Andre Christopher Kirslis, "The SAGA Editor: A Language-Oriented Editor Based on an Incremental LR(1) Parser," PhD Dissertation, University of Illinois at Urbana-Champaign, UIUCDCS-R-85-1236, December 1985.

[68] Jurgen Koenemann & Scott P. Robertson, "Expert Problem Solving Strategies for Program Comprehension," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, New Orleans, Louisiana (1991).

[69] Glenn E. Krasner & Stephen T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.," *Journal of Object-Oriented Programming* (August/September 1988).

[70] Butler W. Lampson, "Bravo Users Manual," in *ALTO User's Handbook*, Xerox Palo Alto Research Center, Palo Alto, California, 1978.

[71] Bernard Lang, "On the Usefulness of Syntax Directed Editors," in *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen & Dag H. Wanvik, eds., Lecture Notes in Computer Science #244, Springer Verlag, Berlin, 1986, 47–51.

[72] Henry Ledgard, Andrew Singer & John Whiteside, *Directions in Human Factors for Interactive Systems*, Springer Verlag, Berlin, 1981.

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

[74] Stanley Letovsky, "Cognitive Processes in Program Comprehension," in *Empirical Studies of Programmers*, Elliot Soloway & Sitharama Iyengar, eds., Ablex Publishing, Norwood, New Jersey, 1986, 58–79.

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

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

[77] Allan MacLean, Victoria Bellotti, Richard Young & Thomas Moran, "Reaching Through Analogy: A Design Rationale Perspective on Roles of Analogy," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, New Orleans, Louisiana (1991).

[78] Allan MacLean, Kathy Carter, Lennart Lövstrand & Tom Moran, "User-Tailorable Systems: Pressing the Issues with Buttons," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Seattle, Washington (April 1990).

[79] Robert L. Mack, Clayton H. Lewis & John M. Carroll, "Learning to Use Word Processors: Problems and Prospects," *ACM Transactions on Office Information Systems* 1 (July 1983), 254–271.

[80] Wendy E. Mackay, "Patterns of Sharing Customizable Software," *Proceedings Conference on Computer-Supported Cooperative Work*, Los Angeles, California (October 7-10, 1990).

[81] Jock Mackinlay, "Automating the Design of Graphical Presentations of Relational Information," *ACM Transactions on Graphics* 5 (1986), 110–141.

[82] Larry M. Masinter, "Global Program Analysis in an Interactive Environment," Xerox Palo Alto Research Center, SSL-80-1, Palo Alto, California, January 1980.

[83] Raul Medina-Mora, "Syntax Directed Editing: Towards Integrated Programming Environments," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-81-113, Pittsburgh, Pennsylvania, March 1982.

[84] Raul Medina-Mora & Peter H. Feiler, "An Incremental Programming Environment," *IEEE Transactions on Software Engineering* SE-7 (September 1981), 472–481.

[85] E. A. T. Merks, J. M. Dyck & R. D. Cameron, "Language Design for Program Manipulation," *IEEE Transactions on Software Engineering* 18 (January 1992), 19.

[86] Norman Meyrowitz & Andries van Dam, "Interactive Editing Systems: Parts I and II," *ACM Computing Surveys* 14 (September 1982), 321–415.

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

[88] Sten Minör, "Interacting with Structure-Oriented Editors," *International Journal of Man-Machine Studies* 37 (October 1992), 399–418.

[89] Elham Morcos-Chounet & Alain Conchon, "PPML: A General Formalism to Specify Prettyprinting," *IFIP '86* (April 9, 1986).

[90] 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).

[91] Donald A. Norman, "Design Rules Based on Analyses of Human Error," *Communications of the ACM* 26 (April 1983), 254–258.

[92] Donald A. Norman & Stephen W. Draper, eds., *User Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

[93] David Notkin, "Interactive Structure-Oriented Computing," PhD Dissertation, Computer Science Department, Carnegie-Mellon University, CMU-CS-84-103, Pittsburgh, Pennsylvania, February 1984.

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

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

[96] Paul W. Oman & Curtis R. Cook, "The Book Paradigm for Improved Maintenance," *IEEE Software* 7 (January 1990), 39.

[97] PROCASE Corporation, *SMARTsystem Reference Guide*, Santa Clara, California, 1990.

[98] John Pasalis, *Realize: An Interactive Graphical Data Structure Presentation System*, Computer Science Division, EECS, University of California, Berkeley, December 1992, Master's Thesis.

[99] G. R. Perkins, R. W. Norman & S. Danicic, "Coping with Deeply Nested Control Structures," *SIGPLAN Notices* 22 (February 1987), 68–77.

[100] Jef Raskin, "Systemic Implications of Leap and an Improved Two-part Cursor: A Case Study," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Austin, Texas (May 1989).

[101] Steven P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 19 (May 1984), 30–41.

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

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

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

[105] Thomas Reps, Tim Teitelbaum & Alan Demers, "Incremental Context Dependent Analysis for Language Based Editors," *ACM Transactions on Programming Languages and Systems* 5 (July 1983), 449–477.

[106] Charles Rich & Richard C. Waters, "The Programmer's Apprentice: A Research Overview," *Computer* 21 (November 1988), 11–25.

[107] Robert S. Rist, "Plans in Programming: Definition, Demonstration, and Development," in *Empirical Studies of Programmers*, Elliot Soloway & Sitharama Iyengar, eds., Ablex Publishing, Norwood, New Jersey, 1986, 28–45.

[108] Caroline Rose, *Inside Macintosh*, Addison Wesley, Reading, Massachusetts, 1985.

[109] Robert W. Scheifler & Jim Gettys, "The X Window System," *ACM Transactions on Graphics* 5 (April 1986), 79–109.

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

[111] Mayer D. Schwartz, Norman M. Delisle & Vimal S. Begwani, "Incremental Compilation in Magpie," *Proceedings of the ACM-SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices* 19 (June 1984).

[112] Jeffrey Alan Scofield, "Editing as a Paradigm for User Interaction," PhD Dissertation, University of Washington Department of Computer Science, 85-08-10, August 1985.

[113] Andrea A. diSessa, "Models of Computation," in *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman & S. W. Draper, eds., Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986, 201–218.

[114] Mary Shaw, "Beyond Programming-in-the-Large: The Next Challenges for Software Engineering," in *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen & Dag H. Wanvik, eds., Lecture Notes in Computer Science #244, Springer Verlag, Berlin, 1986, 519–535.

[115] Mary Shaw, "An Input-Output Model for Interactive Systems," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts (April 1986).

[116] Louis D. Silverstein, "Human Factors for Color Display Systems: Concepts, Methods, and Research," in *Color and the Computer*, H. John Durrett, ed., 1987, 27–61.

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

[118] Barbara Smith & Gerald Kelleher, eds., *Reason Maintenance Systems and Their Applications*, Series in Artificial Intelligence, Ellis Norwood Limited, Chichester, 1988.

[119] Scott R. Smith, David T. Barnard & Ian A. Macleod, "Holophrasted Displays in an Interactive Environment," *International Journal of Man-Machine Studies* 20 (April 1984), 343–355.

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

[121] Richard Stallman, "GNU Emacs Manual," Fifth Edition, Emacs Version 18 for Unix Users, October 1986.

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

[123] Ola Strömfors, "Editing Large Programs Using a Structure-Oriented Editor," in *Advanced Programming Environments*, Reidar Conradi, Tor M. Didriksen & Dag H. Wanvik, eds., Lecture Notes in Computer Science #244, Springer Verlag, Berlin, 1986, 39–46.

[124] Deborah G. Tatar, Gregg Foster & Daniel G. Bobrow, "Design for Conversation: Lessons from Cognoter," *International Journal of Man-Machine Studies* 34 (1991), 185–209.

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

[126] Tim Teitelbaum, Thomas Reps & Susan Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 8–16.

[127] Warren Teitelman, "A Tour Through Cedar," *IEEE Transactions on Software Engineering* SE-11 (March 1985), 285–302.

[128] Larry Tesler, "The Smalltalk Environment," *BYTE* 6 (August 1981), 90–147.

[129] Bruce Tognazzini, *Tog on Interface*, Addison Wesley, Reading, Massachusetts, 1992.

[130] Michael L. Van De Vanter, "Debugging and Error Management in Pan I," Computer Science Division, EECS, University of California, Berkeley, 89/554, December 1989.

[131] Michael L. Van De Vanter, "The Text Module in Pan I," Computer Science Division, EECS, University of California, Berkeley, PIPER Working Paper 90-2, May 1991.

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

[133] Neff Walker & Judith Reitman Olson, "Designing Keybindings to be Easy to Learn and Resistant to Forgetting Even When the Set of Commands is Large," *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Washington, DC (May 1988).

[134] R. C. Waters, "Program editors should not abandon text oriented commands," *SIGPLAN Notices* 17 (1982), 39–46.

[135] Richard C. Waters, "The programmers apprentice: a session with KBEmacs," *IEEE Transactions on Software Engineering* SE-11 (November 1985), 1296–1320.

[136] Mark N. Wegman & Cyril N. Alberga, "Parsing for structural editors (part II)," IBM Corporation, Research Division, RC 9197, Yorktown Heights, NY, January 6, 1982, Extended abstract.

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

[138] Jim Welsh & Mark Toleman, "Conceptual Issues in Language-Based Editor Design," *International Journal of Man-Machine Studies* 37 (October 1992), 419–430.

[139] Terry Winograd, "Beyond Programming Languages," *Communications of the ACM* 22 (July 1979), 391–401.

[140] Steven R. Wood, "Z–The 95 Percent Program Editor," *Proceedings of the ACM-SIGPLAN SIGOA Symposium on Text Manipulation, SIGPLAN Notices* 16 (June 8-10 1981), 1–7.

[141] Daniel M. Yellin & Robert E. Strom, "INC: A Language for Incremental Computations," *ACM Transactions on Programming Languages and Systems* 13 (April 1991), 211.

Appendix A

Annotated View Style Specification for Modula2

;;; Create a new view style called "modula2," to be based on the standard Ladle *and*
;;; Colander *specifications for modula2 (not shown). This view style specification is*
;;; written in a provisional language described more fully in Appendices B and C.

```
(lang:Define-Language-View-Style "modula2"
```

;; Add to the new view style a default configuration for many language-based features:
;; menus and key bindings for language-based operations, standard panel flags, and default
;; fonts and colors. This keeps the specification small and promotes a uniform look and feel
;; across view styles. All of the standard text-based behavior and configuration is
;; inherited unchanged from the global configuration context.

```
  (:add-defaults)
```

;; Build up the basic menu of services, named after this view style, by combining standard
;; parts, following conventions for all view styles. Add extra features, for example
;; automatically generated menus for controlling highlighters and operand levels, as well as
;; special views that are supported by the underlying Colander *specification.*

```
  (:default-menu
    (:menu "Modula2"
          (user::global-clipboard-menu)
          (user::global-textedit-menu)
          (user::global-undo-menu)
          (user::global-store-menu)
          ((:copy-menu user::global-language-menu "Language"
                  ((:level-menu) "Levels")
                  ((:highlighter-menu) "Highlighters")))
          ((:copy-menu user::global-names-menu "Modula2 Names"
                  (lang:Visit-Cross-Ref-View "Visit Cross Ref. View")))
          ((:copy-menu user::language-view-menu "View"
                  (lang:Visit-Cross-Ref-View "Names Cross Reference" 3)
                  (lang:Visit-TOC-View "Proc. Table of Contents" 4)))
          (user::global-text-window-menu "Window")
          (user::global-lang-help-menu)
          ))
```

```
(:define-operand-classes

  ;; Some operand classes, for example "Language Error," are predefined and can be used in
  ;; any view style. Define here a basic set of view-style-specific operand classes to drive
  ;; standard services. The first few are purely syntactic; each is defined by a set of tree
  ;; operator names for the Ladle internal representation, taken from the associated
  ;; Ladle specification for modula2.

  (modula2-proc-opclass
    :title              "Procedure"
    :documentation      "Modula2 procedures."

    ;; For this view style, don't distinguish among procedures, functions, and modules.

    :definition         (:operators "procedure_declaration"
                                    "function_declaration"
                                    "module_declaration")

    ;; Declare that this is a "structural" operand class (the ordinary case), and that all
    ;; the default language-based commands should be made available through the operand
    ;; level command dispatch mechanism.

    :bind-structural-commands

    ;; Use a standard operand menu available that's titled "Procedure"

    :operand-menu       user::procedure-oplevel-menu
    :before             lang:Analyze-Quietly)

  (modula2-decl-opclass
    :title              "Declaration"
    :documentation      "Modula2 declarations."
    :definition         (:operators "declaration"
                                    "definition")
    :bind-structural-commands
    :operand-menu       user::declaration-oplevel-menu
    :before             lang:Analyze-Quietly)

  (modula2-stmt-opclass
    :title              "Statement"
    :documentation      "Modula2 statements."
    :definition         (:operators "stmt")
    :bind-structural-commands
    :operand-menu       user::statement-oplevel-menu
    :before             lang:Analyze-Quietly)

  (modula2-expr-opclass
    :title              "Expression"
    :documentation      "Modula2 expressions."
    :definition         (:operators "expr")
    :bind-structural-commands
    :operand-menu       user::expression-oplevel-menu
    :before             lang:Analyze-Quietly)
;; The next few operand class definitions serve to parameterize standard services that deal
;; with named entities in programming languages. For example, the first specifies what
;; structural components correspond to uses of names in the language. Although they could be
;; used to define operand levels, for supporting language-based navigation and the like, they
;; are not judged to be useful for that purpose.

 (modula2-id-opclass
   :title               "Identifier Instance"
   :documentation       "Lexemes - occurrences of identifiers."
   :definition          (:operators "id")
   :set-command         nil
   :before              lang:Analyze-Quietly)

 ;; Class definition to support Identifier Cross Reference.

 (modula2-decl-id-opclass
   :title               "Id Declaration"
   :documentation       "Lexemes - declarations of identifiers."
   :definition          (:and
                          (:operators "id")
                          (:grandparent-operators "var_declaration"))
   :set-command         nil
   :before              lang:Analyze-Quietly)

 ;; Class definition to support Procedure Table of Contents.

 (modula2-header-opclass
   :title               "Procedure/Function Header"
   :documentation       "Procedure name and argument list"
   :definition          (:operators "procedure_heading"
                                    "function_heading")
   :set-command         nil
   :before              lang:Analyze-Quietly)
 )

;; Parameterize various language-based services for this view style, using view-style-specific
;; operand class definitions (see above).

(:option-values
  lang:Name-Instance-Opclass-Name       modula2-id-opclass
  lang:Name-Decl-Opclass-Name           modula2-decl-id-opclass
  lang:TOC-Header-Opclass-Name          modula2-header-opclass)
```

```
;; Define the menu of operand levels available to users in this view style, specified in terms
;; of both generic and view-style-specific operand class definitions. Add keystroke bindings as
;; accelerators for selecting them quickly, following general and view-style specific conventions.

(:operand-levels        (text:character-level   "^c c")
                        (text:word-level        "^c w")
                        (text:line-level        "^c l")
                        (lang:note-opclass      "^c n")
                        (lang:lexeme-opclass    "^c x")
                        (modula2-expr-opclass   "^C e")
                        (modula2-stmt-opclass   "^C s")
                        (modula2-decl-opclass   "^C d")
                        (modula2-proc-opclass   "^C p")
                        (lang:query-opclass     "^C q")
                        (lang:lang-err-opclass  "^C !" "^C #"))

;; Some highlighters, for example one for the class "Language Error," are predefined and
;; can be used in any view style. Define here view-style-specific highlighters.

(:define-highlighters

  ;; Create a highlighter for procedure headers that uses the color specified in the
  ;; third slot of the foreground color map for the view style. This highlighter is gracious
  ;; (not strict), meaning that it continues to operate (in approximate mode) during periods
  ;; when text and derived structure are inconsistent; during inconsistent periods, however,
  ;; the highlighter display color is taken from the alternate foreground color map for the
  ;; view style, which should by convention be a similar but less saturated version of its
  ;; counterpart in the primary color map.

  (modula2-header-highlighter
    :opclass          modula2-header-opclass
    :title "Function & Procedure Headers"
    :documentation    "An example using color to set off procedures."
    :apropos          (function procedure)
    :effect           :fg3
    :strict?          nil))

;; Define the collection of highlighters available to users in this view style, combining
;; some generic predefined ones with the view-style-specific one defined above. Specify
;; whether each is to be initially on or off. Add keystroke bindings as accelerators for
;; toggling them quickly, following general and view-style specific conventions. Do not
;; permit users to turn off highlighting of the structure cursor in this view style.

(:highlighters          (lang:lang-err-highlighter :on   "^C ^C !")
                        (lang:query-highlighter    :on   "^C ^C q")
                        (modula2-header-highlighter :on  "^C ^C p")
                        (lang:note-highlighter     :off  "^C ^C n")
                        (lang:cursor-highlighter   :on))

;; Configure the appearance of window panels for the view style, beginning with.
;; a view style logo.

(:option-values
  win:Panel-View-Logo-Bitmap           "mod2-logo.29x17")
```

```
;; Add panel flags to the default set inherited from the global configuration. The default
;; set includes one that appears when the view permits no editing, one that indicates unsaved
;; modifications, and one that announces when automatic text filling is in effect. The flags
;; added here are specified using scoped variables (whose local values dynamically define
;; whether the flag is on or off), and standard bitmaps and colors for a uniform look and
;; feel across view styles.

(:add-flags
  (user:%pad-flag-variable%           user::Space-Bitmap)
  (otree:Parse-Is-Current?              (user::Parse-Current-Bitmap
                                         user::Panel-Flag-Green)
                                        (user::Parse-Not-Current-Bitmap
                                         user::Panel-Flag-Pale-Red))
  (sem:Database-Updated?                (user::Database-Updated-Bitmap
                                         user::Panel-Flag-Green)
                                        (user::Database-Not-Updated-Bitmap
                                         user::Panel-Flag-Pale-Red))
  (lang:Language-Errors-Present?        (user::Language-Errors-Present-Bitmap
                                         user::Panel-Flag-Red)))

;; Configure text rendering for the view style.

(:option-values

  ;; Assign font selections to the font map for the view style. There are eight slots but only
  ;; five are used by the current implementation. Font assignments are made by the lexer as
  ;; follows: (0) newly entered, unanalyzed text; (1) scanned but not parsed (this slot is only
  ;; effective during debugging); (2) language keywords; (3) comments; and (4) language
  ;; identifiers.

  twin:Text-Window-Fontmap
                ("-*-lucidatypewriter-medium-r-*-*-12-*-*-*-*-*-*-*"
                 "-*-lucida-medium-r-*-*-12-*-*-*-*-*-*-*"
                 "-*-lucidabright-medium-r-*-*-12-*-*-*-*-*-*-*"
                 "-*-new century schoolbook-medium-r-*-*-12-*-*-*-*-*-*-*"
                 "-*-lucidabright-demibold-r-*-*-12-*-*-*-*-*-*-*")

  ;; Assign font selections to the alternate font map for the view style. A single, global
  ;; "demo" switch (option variable) controls whether the demo fontmaps are used.

  twin:Text-Window-Demo-Fontmap
                ("-*-lucidatypewriter-medium-r-*-*-17-*-*-*-*-*-*-*"
                 "-*-lucida-medium-r-*-*-18-*-*-*-*-*-*-*"
                 "-*-lucidabright-medium-r-*-*-18-*-*-*-*-*-*-*"
                 "-*-lucidabright-medium-r-*-*-14-*-*-*-*-*-*-*"
                 "-*-lucidabright-demibold-r-*-*-18-*-*-*-*-*-*-*")

  ;; Specify default (initial) size for newly created text windows in the view style, computed
  ;; relative to the nominal width and height of the default (font map slot zero) font for the
  ;; view style.

  twin:Text-Window-Init-Cols          60
  twin:Text-Window-Init-Rows          25
```

```
    ;; Render text using a tabwidth computed in pixels as 5 times the nominal width of the
    ;; default (font map slot zero) font for the view style. This width is used for all fonts in
    ;; the view style so tab alignment may be obtained in the presence of multiple,
    ;; proportionally-spaced fonts.

    twin:Text-Window-Tabwidth              5

    ;; Assign color selections to the foreground (ink) color map for the view style. There are
    ;; four slots corresponding to possible ink selections, with the default (the first) color
    ;; being black by convention. Each slot contains a pair of color specifications: the left
    ;; side is used for color monitors and the right side for monochrome monitors. These color
    ;; specifications are normally inherited unchanged from the global defaults in order to
    ;; promote a uniform look and feel.

    twin:Text-Window-Fg-Colormap           (("black" . "black")
                                            ("IndianRed1" . "black")
                                            ("SlateBlue3" . "black")
                                            ("turquoise3" . "black"))

    ;; Assign color selections to the background (shading) color map for the view style. There
    ;; are three slots corresponding to three possible shadings, where the background is assumed
    ;; to be white when no shading is in effect.

    twin:Text-Window-Bg-Colormap           (("LightYellow1" . "white")
                                            ("alice blue" . "white")
                                            ("lavender" . "white"))

  ;; Assign color selection to alternate versions of the two color maps: foreground and
  ;; background. These alternate versions are used during periods when text and derived
  ;; language-based information are inconsistent. User interface design convention dictates
  ;; that these colors be chosen similar in hue to their counterparts in primary color maps,
  ;; but with reduced saturation to suggest metaphorically that the information they represent
  ;; is not strictly reliable.

    twin:Text-Window-Fg-Alt-Colormap       (("black" . "black")
                                            ("IndianRed3" . "black")
                                            ("SkyBlue2" . "black")
                                            ("turquoise4" . "black"))

    twin:Text-Window-Bg-Alt-Colormap       (("ivory" . "white")
                                            ("azure" . "white")
                                            ("lavender blush" . "white"))

  ;; There is also "demo" version for each of the above four color maps (now shown here).
  ;; Color assignments suitable for the single user don't show up at a distance, so demo colors
  ;; are typically brighter and more saturated.

    ))

;;; This concludes the view style specification for the view style named "modula2." Any of
;;; the many other configuration options in Pan may also be assigned values in a view
;;; style specification. Templates may be specified to add syntax-directed style of editing
;;; to standard language-based commands. New view-style-specific commands may be defined and
;;; bound to keystrokes or menus. Finally, procedures may be attached to various "hooks"
;;; run at predefined events (for example at view initialization and after each analysis).
```

Appendix B

Language-Based View Style Specification

A language-based view style in *Pan* is a design for interaction between some intended category of users, performing some collection of tasks, and their software documents expressed in some formal language notation. This appendix summarizes how such a design can be recorded in *Pan*'s provisional specification language.

Language-based view styles are constructed in isolation from the main *Pan* system and from each other. The system has (almost) no information concerning the name, the location, or even the existence of any view style until it is loaded. View styles (and in particular the *Ladle* and *Colander* descriptions they use) can be loaded at any time, as many as desired, as long as they do not conflict in name. Each contains all specifications necessary to enable the system to provide a wide variety of language-based services in the target language, including per-language control of almost every aspect of user interaction. A sample specification for Modula-2 appears in Appendix A.

B.1 Associating View Styles with File Types

View style configuration in *Pan* is arranged so that the system contains as little information as possible about view styles not yet loaded. Something must be known, however, so that a view style can be located and loaded when needed. A declaration of the following form supplies the needed information for *Pan*'s file system interface:

```
(File-Type-Use-Language-View-Style "col" "colander")
```

This specifies that any file whose name ends with the extension "`.col`," will be visited by *Pan* in the context of the view style named "`colander`." *Pan*'s default configuration file contains similar declarations for standard file extensions, but they may be overridden and extended at any time. Should *Pan* be extended (as its design permits) with an interface to another kind of persistent store, a similar declaration would map types (in whatever sense supported by the store) to view styles.

Personal customization code may alter a standard language-based view style configuration by changing the values of options and other bindings. For example, in a file named `mlv-colander-mode.cl` the following form would request that a customization file be loaded immediately after the loading of view style "`colander`" (or at editor startup time if the view style is preloaded).

```
(Add-Language-View-Style-Auto-Require "colander" "mlv-colander-module")
```

The customization file `mlv-colander-module.cl` might then override configurations with the following form. The characters "`<>{}*`" are metacharacters in this and all other examples in this section.

```
(With-View-Style-Scope (view-style colander-view-style)
  ...<set options> )
```

B.2 The View Style Specification File

A language-based view style specification resides in a file named "`<view style>-lang`" located somewhere on the system's search path, where "`<view style>`" is the view style's name. In the provisional specification mechanism, the specification file is loaded directly by COMMON LISP, so it may contain mix of traditional *Pan* extension code and declarative specifications in S-expression form. The aim, of course, is to minimize the amount of auxiliary procedural code that is needed for routine cases, but to retain flexibility to prototype new variations that cannot be expressed (yet) by the specification language. Eventually, auxiliary COMMON LISP code will reside in separate COMMON LISP source files, just as they do now for *Colander* descriptions.

B.3 The `Define-Language-View-Style` Form

The declarative part of a view style configuration file consists of S-expressions in the provisional specification language. This syntax amounts to the intermediate form of a specification language that has not yet been implemented. These expressions are wrapped in the top-level form `Define-Language-View-Style`, a COMMON LISP macro that processes the specifications. Building the intermediate form first permits early development of the back-end mechanisms necessary to process the specifications. This intermediate form is analogous to *Colander*'s PLI intermediate form, as produced by the *Colander* preprocessor.

In this intermediate language, a complete and workable *Pan* language-based view style is defined by an instance of following form:

```
(Define-Language-View-Style <name>
  {<specifier>}* )
```

The string "`<name>`" identifies the language; the detailed nature of a "`<specifier>`" will be described below. A complete example for Modula-2 appears in Appendix A. Loading a file containing this top level form has several effects:

1. it loads *Ladle* and *Colander* descriptions, using conventions for names and locations of files;
2. it creates a new view style (a configuration binding scope); and
3. it configures the newly created view style according to the list of specifiers.

As always with *Pan* scoping, considerable default behavior may be inherited from the global scope.

B.4 View Styles, Object Classes, and Scopes

The top-level form `Define-Language-View-Style` automatically creates a view style named "`<name>-view-style`," where "`<name>`" is the view style name supplied to the form. The existence and name of this view style may be relevant to end users and designers (see above how one might add personal language-specific customizations). Nothing else in this section needs to be exposed to end users or designers.

Loading the `Define-Language-View-Style` form also also creates new language-specific subclasses (in the CLOS sense) of edit objects and views. These are part of the implementation, but they generally need not be exposed to designers. These are:

- a new edit-object class named "`$<name>-edit-object$`"
- a new view class named "`$<name>-text-view$`"

where "`<name>`" is the view style name supplied to the form. These classes are subclasses of `$lb-edit-object$` and `$lb-text-view$` respectively, both of which carry methods for generic language-based functionality (even though there will never be instances of the two superclasses).

Figure 4.4 on page 45 shows *Pan*'s edit object class hierarchy, and Figure 4.5 on page 47 the hierarchy for view classes.

Loading the `Define-Language-View-Style` form additionally:

1. establishes an internal map between the file type (e.g. "`col`") and the edit-object class for visited files of that type;
2. makes `<name>-view-style` the default view style for all visited files of that type.

B.5 View Style Specifiers

A "specifier" in the intermediate specification language is a COMMON LISP form (list) that begins with an identifying keyword and whose subsequent arguments are suitable to the particular specifier. Arguments may be symbols (unquoted names), COMMON LISP data literals, lists of arguments, and keywords. For example, the specification fragment in Figure B.1 establishes simple key bindings for the view style; the fragment in Figure B.2

```
(:bind-keys
 Set-Oplevel-To-Language-Error          "^C !"
 Set-Oplevel-To-Query                   "^C q")
```

Figure B.1: Key Binding Specification

sets option values in the view style; the fragment in Figure B.3 defines an operand class (and level, partially) for the "Colander" view style; and the fragment in Figure B.4 specifies the standard menu for the "Colander" view style.

Specifiers currently defined appear in Table B.1; see Appendix C for documentation on each.

```
 (:option-values
  Text-Window-Tabwidth                   7
  Announce-Error-Counts-After-Analysis   nil)
```

Figure B.2: Option Value Specification

```
(:define-operand-classes
 (colander-decl-opclass
    :title         "Declaration"
    :documentation "Colander declarations."
    :definition    (:operators "fact_decl"
                               "entity_property_decl"
                               "maintained_property_decl"
                               "node_property_decl"
                               "entity_name_decl"
                               "datapool_name_decl"
                               "pure_decl"
                               "mode_decl")
    :bind-structural-commands
    ;; Use a standard operand menu titled "Declaration"
    :operand-menu        user::declaration-oplevel-menu
    :before              lang:Analyze-Quietly))
```

Figure B.3: Opclass Specification for "Colander Declarations"

```
(:default-menu
 (:copy-menu user::default-language-menu "Colander"
            ((:copy-menu user::global-language-menu "Language"
                         ((:level-menu "Levels"))
                         ((:highlighter-menu "Highlighters")))
             nil 4)
            ((:menu "Preprocess"
                    (Save-Write-PLI-File "Save/Write-PLI File")
                    (Write-PLI-File "Write-PLI File"))
             nil 5)))
```

Figure B.4: Menu Specification for "Colander"

```
(:add-char-matches {<left-char> <right-char>}* )
(:add-char-sets {<char-set-name> <char-list>}* )
(:add-defaults)
(:add-flags {<flag-descriptor>}* )
(:after-analysis-function <function> )
(:before-analysis-function <function> )
(:bind-keys {<command> <key-descriptor>}* )
(:default-menu <menu-spec> )
(:define-highlighters {<highlighter-description>}* )
(:define-operand-classes {<class-description>}* )
(:delete-char-matches {<left-char> <right-char>}* )
(:delete-char-sets {<char-set-name> <char-list>}* )
(:delete-flags {<flag-descriptor>}* )
(:highlighters {(<highlighter-name> <:on|:off> <key-spec>* )}* )
(:init-function <function> )
(:operand-levels {(<operand-class-name> <key-spec>* )}* )
(:option-values {<option-name> <value>}* )
(:sde-oplevel-menus <oplevel-menu-name> {<opl-menu-spec>}* )
(:shadow-char-matches {<left-char> <right-char>}* )
(:shadow-char-sets {<char-set-name> <char-list>}* )
(:shadow-flags {<flag-descriptor>}* )
```

Table B.1: Language-Based View Style Specifiers

```
(setf (variable ...)  ..)
Add-Default-Language-Configuration-To-View-Style
Bind-Key
Bind-Menu
Bind-Oplevel-Command
Bind-Oplevel-Menu
Copy-Menu
Create-Menu
Define-Highlighter
Define-Operand-Class
Define-Structural-Oplevel-Commands
Set-Char-Match
Set-Char-Set
Window-Modify-Flag-Collection
```
and some view methods

Table B.2: Configuration Forms That Implement Specifiers

B.6 Specifier Implementation

Specifiers are implemented by macro expansion into appropriate calls on existing *Pan* configuration mechanisms, including those listed in Table B.2.[1]

The implementation of `Define-Language-View-Style` ensures that all of these are evaluated in the proper context: in the scope of the new view style so that bindings will be added correctly, and with the *Ladle* global syntactic state to the newly loaded language so that operator names in the *Ladle* description may be named and processed correctly.

B.7 Creating New Specifiers

The implementation of `Define-Language-View-Style` defines no specifiers directly; all are created dynamically via calls to a defining form, so more can be added at any time. This feature is of special value during development of the intermediate language and its supporting mechanisms, but it might also be of use by designers and users.

For example, the experimental module for syntax directed editing (module `sde.cl`) is normally loaded from the library only when needed (possibly at runtime), typically along with the first language-based view style that uses it. When loaded, the `sde` module extends the view style specification language by adding the new specifier `:sde-oplevel-menus`. A client view style need only ensure (via a COMMON LISP `require`) that the `sde` module be loaded before attempting using this specifier to create menus that support placeholder expansion templates.

It is not clear whether this degree of flexibility can persist usefully after the addition of a front-end syntax with preprocessing.

[1]In many cases the internal equivalent of the configuration form is used.

For example, an unusually simple definition (one with no argument checking at all) is:[2]

```
(def-language-description-specifier :add-flags
  "(:add-flags {<flag-descriptor>}* )
Add all specified flags to the flag collection in the view
style of the language. See Window-Modify-Flag-Collection."
  #'(lambda (arglist)
      '((win:Window-Modify-Flag-Collection :view-style
                                           :add
                                           ',arglist))))
```

This translates a specification of the form

```
(:add-flags <flag specs>)
```

into

```
(win:Window-Modify-Flag-Collection :view-style
                                   :add
                                   <flag specs>)
```

which is how an equivalent specification for some other view style might appear in the standard configuration file `default.cl`.

A slightly more complex definition is:

```
(def-language-description-specifier :bind-keys
  "(:bind-keys {<command> <key-descriptor>}* )
For each pair in list, bind <command> to the key sequence
<key-descriptor> in the view style of the language.
See Bind-Key."
  #'(lambda (arglist)
      (do ((in-forms arglist (cddr in-forms))
           (out-forms (create-empty-queue)))
          ;; loop termination
          ((or (null in-forms)
               (length=1 in-forms))
           (when (length=1 in-forms)
             (Editor-Warn ":bind-keys ignoring odd argument ~S"
                          in-forms))
           (list-queue-elements out-forms))
          ;; loop body
          (let ((cmd (first in-forms))
                (keyspec (second in-forms)))
            (enqueue '(Bind-Key ',cmd ,keyspec :view-style)
                     out-forms)))))
```

Some specifiers are rather more complex, especially those that accept complex nested specifications. Each implements what amounts to a compiler, along with error diagnosis and recovery, for a "little language."

[2]Note that this definitional form takes a string argument containing documentation; this is incorporated by *Pan*'s online help system.

Appendix C

Language-Based View Style Specifiers

This appendix contains documentation that was generated automatically from information collected by `def-language-description-specifier`. Similar information is available in *Pan*'s help view (see the "Documentation" submenu).

In many cases, the documentation here refers the reader to documentation for other definitional forms in the language. These are available in *Pan*'s online documentation, and the services are described in more detail the *Pan* user manual [31]. The characters "<>{}*" are metacharacters in this documentation.

`(:add-char-matches {<left-char> <right-char>}* )`

For each pair of characters, add their definition as a matching pair in the view style of the language. See `Set-Char-Match` for more documentation about character matching.

`(:add-char-sets {<char-set-name> <char-list>}* )`

For each pair, add every character in `<char-list>` to the character set `<char-set-name>` in the view style of the language. See `Set-Char-Set` for more documentation about character sets.

`(:add-defaults)`

Adds the default bindings for general language based editing to the view-style of the language. These may be overridden piecewise by subsequent specifications.

`(:add-flags {<flag-descriptor>}* )`

Add all specified flags to the flag collection in the view style of the language. See `Window-Modify-Flag-Collection` for more documentation about panel flags.

`(:after-analysis-function <function> )`

Add a view method function to be run after each analysis; it is called in the full context of `<view>`. Note that the definition of this function will not be evaluated with the new language syntax current, so it should contain no load-time grammar transformations. See defgeneric of `view-notify-after-analysis` for more documentation on this generic function.

`(:before-analysis-function <function> )`

Add a view method function (of one argument `<view>`) to be run before each analysis; it is called in the full context of `<view>`. Note that the definition of this function will be not be evaluated with the new language syntax current, so it should contain no load-time grammar transformations. See defgeneric of `view-notify-before-analysis` for more documentation on this generic function.

`(:bind-keys {<command> <key-descriptor>}* )`

For each pair in list, bind `<command>` to the key sequence `<key-descriptor>` in the view style of the language. See `Bind-Key` for more documentation about key bindings.

`(:default-menu <menu-spec> )`

Use the menu described by `<menu-spec>` as the default menu in the view style of the language. `<menu-spec>` may take several forms:

- `<menu-name>`

  A symbol specification causes the named menu, presumed to exist already, to be used.

- `(:menu <title> { (<entry> [<label>]) }* )`

  A newly created menu will be used, to appear with `<title>`, a string, at the top. Any number of bindings may be follow, each of which will be added in order to the newly created menu. `<entry>` may be one of:

  - the name of a bindable editor command,
  - the name of another menu to be added as a pullright submenu,
  - a `<menu-spec>` recursively, or
  - the keyword `:non-selectable`.

  The entry will be labeled with the string `<label>`, if given, or a default. See `Bind-Menu` for more documentation on new menu creation.

- `(:copy-menu <menu-name><title> { (<entry> [<label> [<position>]]) }* )`

  A copy will be made of the menu, presumed to exist already, named by the symbol `<menu-name>`, and the copy will appear with `<title>`, a string, at the top. Additions will be made to the new menu as specified by any number of additional bindings. Each binding is specified by `<entry>`, as described above. The entry will be labeled with the string `<label>`, if given, or a default if `<label>` is missing or `nil`. The entry will be inserted at `<position>`, an integer, or at the end of the menu if missing. See `Bind-Menu` for more documentation on menu copying.

- `(:level-menu [<title>])`

  A menu will be constructed, to appear with `<title>`, a string at top; default title is "`Set Levels`". Bindings will automatically be added for each of the current operand levels for which a setting command has been defined. Each such binding will appear as an entry whose title is the title of the operand level, annotated with a key binding, if any, that invokes it in the local context.

- `(:highlighter-menu [<title>])`

  A menu will be constructed, to appear with **`<title>`**, a string at top; default title is "Toggle Highlighters". Bindings will automatically be added for each of the currently available highlighters for which a toggling command has been defined. Each such binding will appear as an entry whose title is the title of the highlighter, annotated with a key binding, if any, that invokes it in the local context.

`(:define-highlighters {<highlighter-description>}* )`

Define a new highlighter for each **`<highlighter-description>`**. A highlighter associates with an operand class a visual effect that, when a highlighter is activated, will be maintained automatically. A `<highlighter-description>` has the form:

```
(<name>
 :opclass <operand-class-name>
 [:title <title>]
 [:documentation> <documentation>]
 [:apropos <apropos>]
 [:effect :fg1|:fg2|:fg3|:bg1|:bg2 ]
 [:toggle-command <toggle-command-name>]
 [:strict? t|nil ]
 )
```

- `<Operand-class-name>` is a symbol naming a previously defined operand class. This class defines which nodes will be highlighted.

- `<Title>` is a string by which the highlighter is described to users, for example, in a menu. The title need not be unique; for example many languages may have a level titled "`Statement`". Default is the name of the operand class.

- `<Documentation>` is a descriptive string.

- **`<Apropos>`** is a symbol or a list of symbols. Apropos is generated for the specified symbols as well as for the name of the highlighter.

- **`<Effect>`** must be one of the specified keywords. The three foreground effects use ink colors 1 through 3 respectively, as configured by option Text-Window-Fg-Colormap. The two background effects use shading colors 1 and 2, as configured by option **`Text-Window-Bg-Colormap`**. Default is `:fg1`.

- **`<Toggle-command>`** specifies the name of the bindable command that toggles the highlighter in current view between active and inactive. The default, when there is no toggle-command specification, is to create automatically a command named **`Toggle-<name>`**. That name may be overridden by this specification, or, if **`<toggle-command>`** is `nil`, the creation of the command may be suppressed entirely.

- `<Strict?>` specifies whether the effect will be cleared when text and analyzed data become inconsistent. Default is `t`.

See `Define-Highlighter` for more documentation on highlighters.

`(:define-operand-classes {<class-description>}* )`

Define a new operand class for each class specification `<class-description>`. An operand class describes a dynamically determined subset of the tree nodes in a document, and associates certain kinds of useful behaviors with operations on those nodes. A `<class-description>` has the form:

```
(<name>
 :definition <test-spec>
 [:title <title>]
 [:documentation> <documentation>]
 [:apropos <apropos>]
 [:extension <extension> ]
 [:set-command <set-command-name>]
 [:bind-structural-commands]
 [:operand-menu <menu-spec>]
 [:operand-menu-command <menu-command>]
 [:before <before-daemon>]
 [:after <after-daemon>]
 )
```

- `<test-spec>` describes the meaning of the operand class by defining a test (i.e. a predicate) for deciding dynamically whether a given node is a member of the class or not.`<test-spec>` is a possibly nested list in one of the following forms:
  - `(:operators {<operator-name>}* )`
    The arguments form a standard *Ladle* phylum specification, where each `<operator-name>` is a string that names an operator in the language. Only nodes whose operator is listed are members.
  - `(:parent-operators {<operator-name>}* )`
    The arguments form a standard *Ladle* phylum specification, where each `<operator-name>` is a string that names an operator in the language. Only nodes whose parent operator is listed are members.
  - `(:grandparent-operators {<operator-name>}* )`
    The arguments form a standard *Ladle* phylum specification, where each `<operator-name>` is a string that names an operator in the language. Only nodes whose grandparent operator is listed are members.
  - `(:predicate <predicate> )`
    `<predicate>` is a function (or the name of a function) taking one argument: a tree node. The function is guaranteed to be run with correct global language state, both syntactic and semantic.
  - `(:member-variable <variable-name> [<test>] )`

`<variable-name>` is the symbol name of a *Pan* variable, as defined by one of `Define-Variable`, `Define-Option-Variable`, or `Defvariable`. Class membership is computed dynamically by evaluating a particular tree node for membership in the list that is the current value of the variable. `<test>` is a function of two arguments used to determine membership, default is `#'eq`.

- `(:or {<test-spec>}* )`
  `(:and {<test-spec>}* )`
  `(:not <test-spec> )`

  `<test-spec>` may be a nested specification. Membership is determined by combining the included specifications into a compound test, using standard Lisp logical operations.

- `<Title>` is a string by which the level is made visible to users, for example, in a menu. The title need not be unique; for example many languages may have a level titled "Statement."

- `<Documentation>` is a descriptive string.

- `<Apropos>` is a symbol or a list of symbols. Apropos is generated for the specified symbols as well as for the name of the level.

- `<Extension>` is a symbol or a list of symbols, each of which names a *Pan* scoped variable, created using `Define-Variable`, `Define-Option-Variable`, or `Defvariable`. This provides an alternate definition of class membership that can makes some internal operations much faster. A single variable name identifies a variable whose value can always be relied upon to be a list of nodes that enumerate the membership of the class; when more than one variable is named, the set union of their values defines class membership. The extensional definition is optional, and when present, no attempt is made to verify equivalence between it and the predicate (`<text-spec>`) definition.

- `<Set-command>` specifies the name of the bindable command that sets the current view's oplevel to this oplevel (a panel menu permits them to be set too). The default, when there is no set-command specification, is to create automatically a command is named `Set-Oplevel-To-<name>`. That name may be overridden by this specification, or, if `<set-command>` is `nil`, the creation of the command may be suppressed entirely.

- The presence of the `:bind-structural-commands` keyword specifies that operand bindings for this level be created for the standard set of structural commands, see `Define-Structural-Oplevel-Commands` for more documentation.

- `<Menu-spec>` describes a menu to bind as the designated operand level menu for this operand class. The specification may be a symbol naming a preexisting menu, or it may be a recursive specification of a new menu, as with the argument to `:default-menu`. See documentation for specifier `:default-menu` (above) and for `Bind-Oplevel-Menu` for more documentation on operand level menu bindings.

- `<Menu-command>` replaces the normal operand menu dispatch command; this causes `<menu-name>` to be ignored unless handled specially by the replacement.

- `<Before-daemon>` is a function of no arguments. It will be called before each execution of an operation dispatched at the new level.
- `<After-daemon>` is a function of no arguments. It will be called after each execution of an operation dispatched at the new level.

See `Define-Operand-Class` for more documentation on operand class definition.

`(:delete-char-matches {<left-char> <right-char>}* )`

For each pair of characters, delete their definition as a matching pair in the view style of the language. See `Set-Char-Match` for more documentation about character matching.

`(:delete-char-sets {<char-set-name> <char-list>}* )`

For each pair, delete every character in `<char-list>` from the character set `<char-set-name>` in the view style of the language. See `Set-Char-Set` for more documentation about character sets.

`(:delete-flags {<flag-descriptor>}* )`

Delete all specified flags from the flag collection in the view style of the language. See `Window-Modify-Flag-Collection` for more documentation about panel flags.

`(:highlighters {(<highlighter-name> <:on|:off> <key-spec>* )}* )`

Specify the names of highlighters to be made available to the user. Each instance of `<operand-class-name>`, a symbol, names a level, and these choices appear in a panel menu. When one more instances of `<key-spec>`, a string containing a keystroke specification, are included, the command that toggles the highlighter is bound to those keystrokes.

`(:init-function <function> )`

Add a view method function (of one argument `<view>`) to be run in the scoping context, not the full context, of each newly created primary view for the language, after all other initialization is done. Note that the definition of this function will be not be evaluated with the new language syntax current, so it should contain no load-time grammar transformations. See defgeneric of `view-initialize` for more documentation on this generic function.

`(:operand-levels {(<operand-class-name> <key-spec>* )}* )`

Specify the names of operand classes to be made available to the user as possible "operand levels," which control some kinds of language-based interaction. Each instance of `<operand-class-name>`, a symbol, names a level, and these choices appear in a panel menu. When one more instances of `<key-spec>`, a string containing a keystroke specification, are included, the command that sets the current level to this class is bound to those keystrokes. Any attempt by users to set the level to an operand class not declared in this way will signal an error.

`(:option-values {<option-name> <value>}* )`

For each pair in list, set `<option-name>` to `<value>` in the view style of the language. See `Define-Option-Variable` for more documentation on *Pan* options.

`(:sde-oplevel-menus <oplevel-menu-name> {<opl-menu-spec>}* )`

Specify a mapping between operators in the grammar and special operand menus for syntax-directed editing with templates. Each operand menu described by an `<opl-menu-spec>` is created automatically by copying the menu named by `<oplevel-menu-name>` and adding a submenu for template expansion appropriate to the operator. Each `<opl-menu-spec>` has the form:
  `(<operator-name> <template-menu-title> {<template-entry>}* )`
where `<operator-name>` is the string name of an operator in the *Ladle* grammar for the language, and `<template-menu-title>` is the string that will appear as the name of the menu containing templates. Each `<template-entry>` has the form:
  `(<template-entry-label> {<template-string-component>}* )`
where `<template-entry-label>` is the string name of the menu entry for invoking the template and the `<template-string-component>` strings make up the template itself. When a template contains more than one string, during expansion a newline will be inserted before each component after the first.

`(:shadow-char-matches {<left-char> <right-char>}* )`

For each pair of characters, shadow their definition as a matching pair in the view style of the language. See `Set-Char-Match` for more documentation about character matching.

`(:shadow-char-sets {<char-set-name> <char-list>}* )`

For each pair, delete every character in `<char-list>` from the character set `<char-set-name>` in the view style of the language. See `Set-Char-Set` for more documentation about character sets.

`(:shadow-flags {<flag-descriptor>}* )`

Shadow all specified flags from the flag collection in the view style of the language. See `Window-Modify-Flag-Collection` for more documentation about panel flags.