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

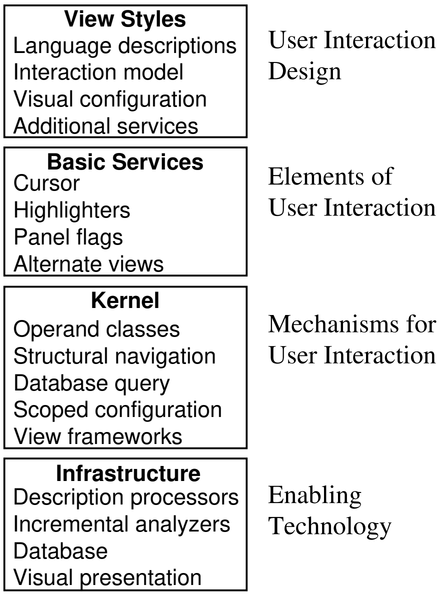

Figure 3.7: Design Layers

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*.

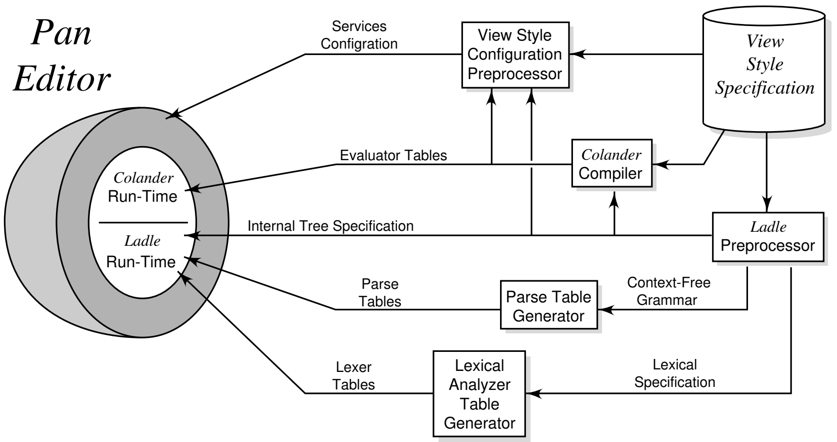

Figure 4.1: Preprocessing Language-Based View Style Specifications

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]**La**nguage **D**escription **L**anguag**e**

[2]**Con**straint **La**nguage **and** Interpret**er**

[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 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.

`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

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.

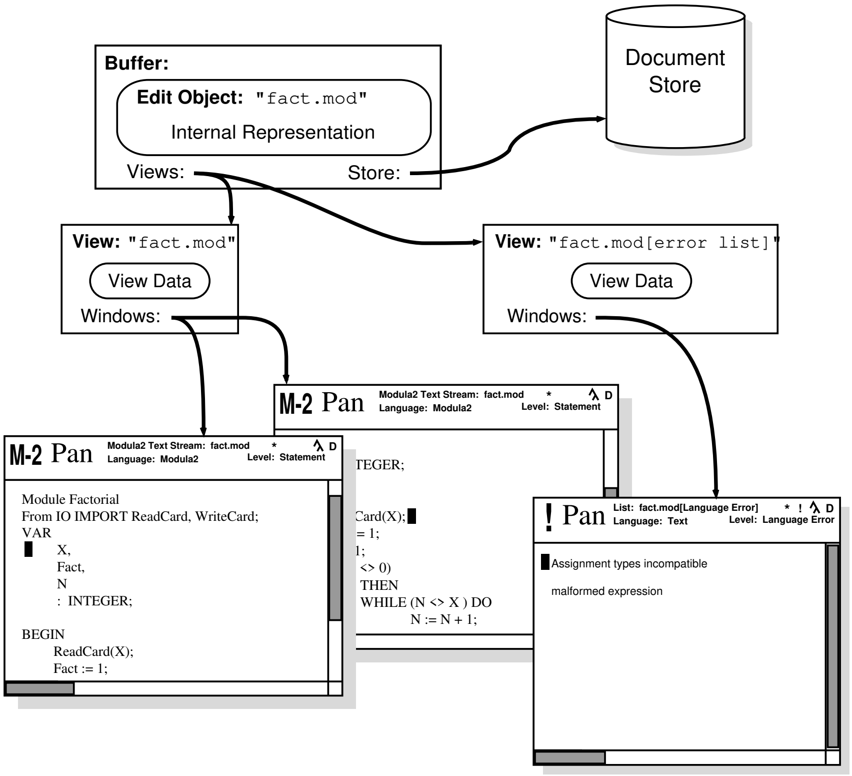

Figure 4.3: Example: Internal Object/View Model

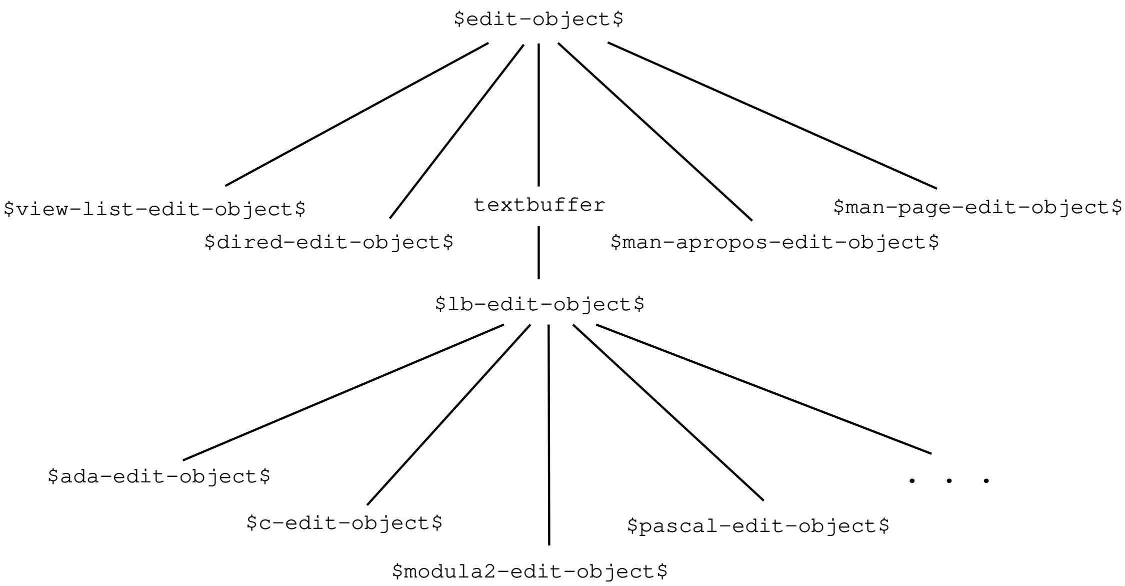

Figure 4.4: Edit Object Class Hierarchy

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

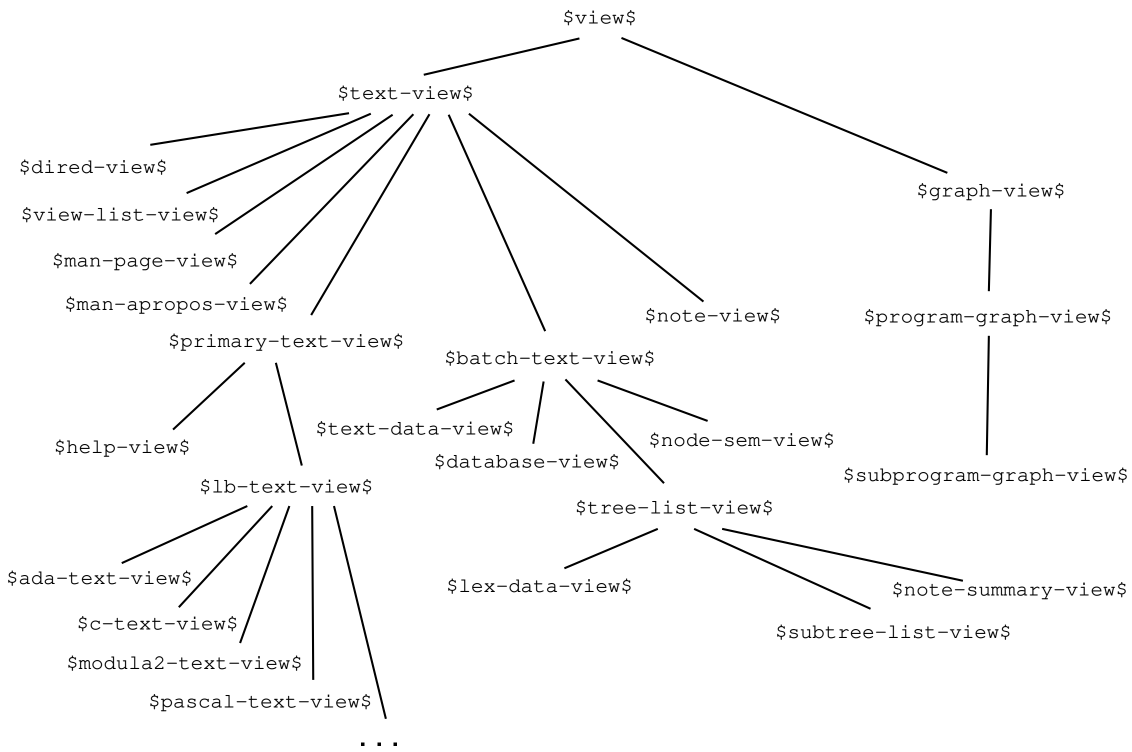

Figure 4.5: View Class Hierarchy

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.

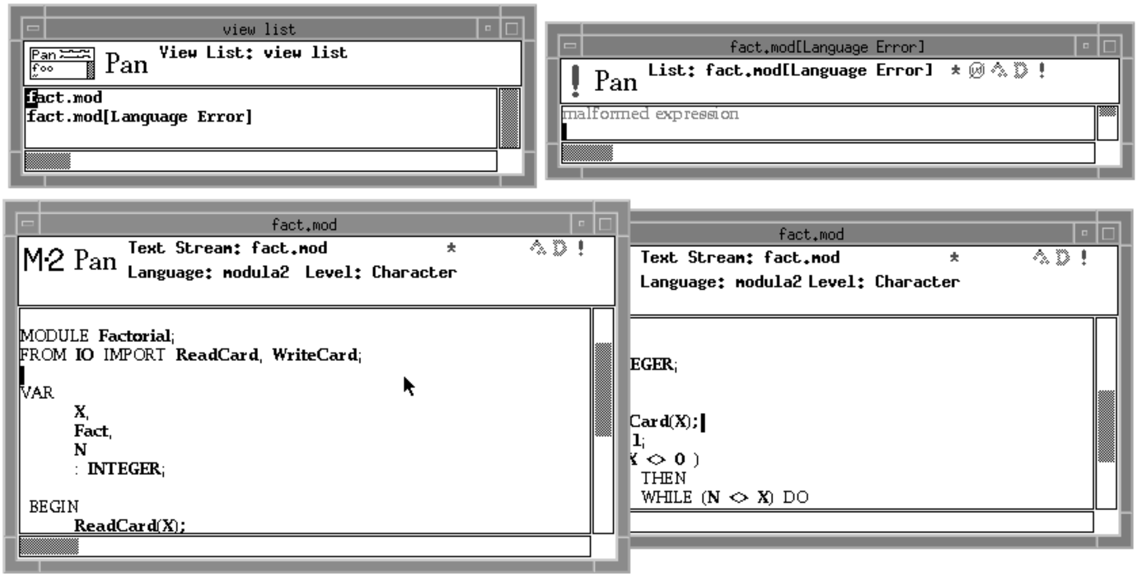

Figure 4.7: Using *Pan*: Multiple Views

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.