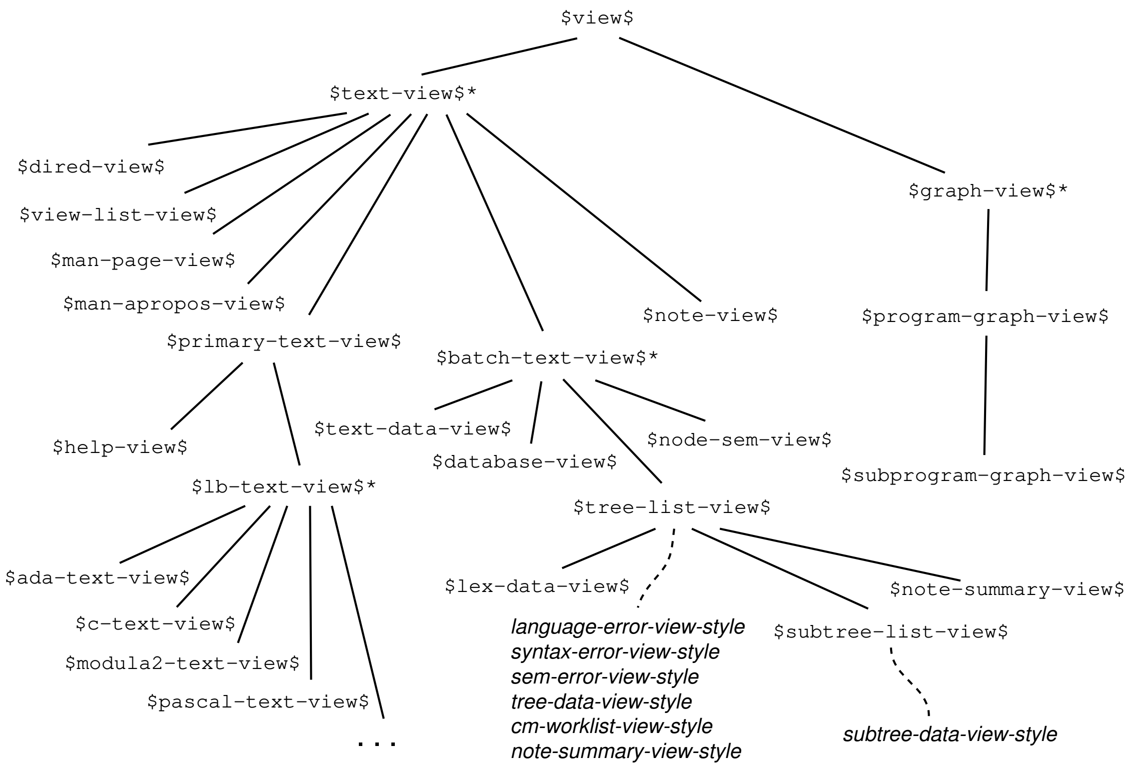

Figure 4.8: View Classes and View Styles

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).

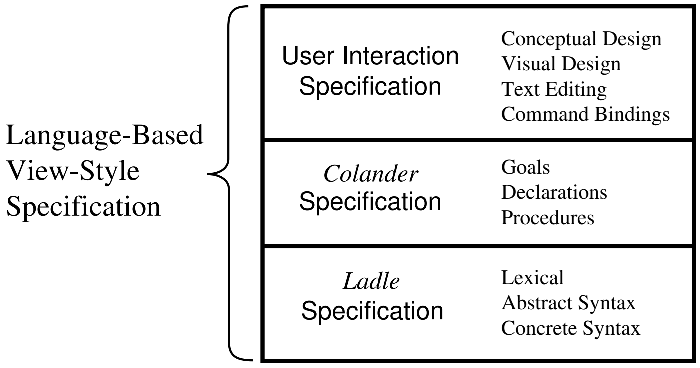

Figure 4.11: Specification Layers

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.

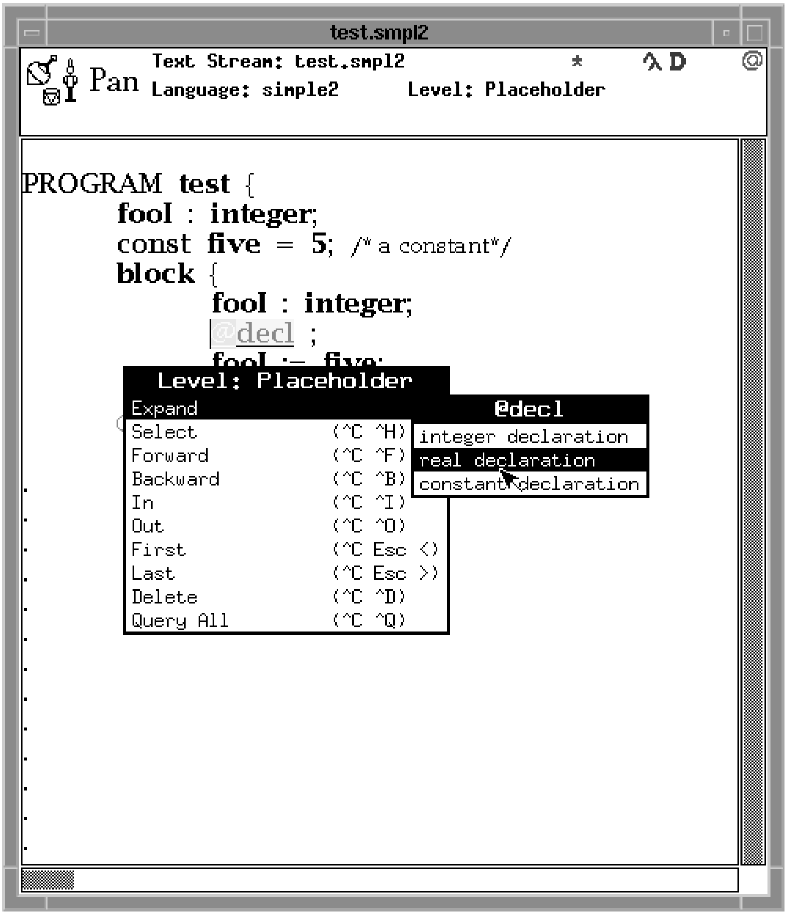

Figure 5.1: Using *Pan*: Expanding a Placeholder