ELSEVIER

Information and Software Technology 44 (2002) 767–782

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

The documentary structure of source code

Michael L. Van De Vanter

Sun Microsystems Laboratories, 4150 Network Circle, UMTV29-112, Santa Clara, CA 95054 USA

Abstract

Many tools designed to help programmers view and manipulate source code exploit the formal structure of the programming language. Language-based tools use information derived via linguistic analysis to offer services that are impractical for purely text-based tools. In order to be effective, however, language-based tools must be designed to account properly for the documentary structure of source code: a structure that is largely orthogonal to the linguistic but no less important. Documentary structure includes, in addition to the language text, all extra-lingual information added by programmers for the sole purpose of aiding the human reader: comments, white space, and choice of names. Largely ignored in the research literature, documentary structure occupies a central role in the practice of programming. An examination of the documentary structure of programs leads to a better understanding of requirements for tool architectures. © 2002 Sun Microsystems Inc. Published by Elsevier Science B.V. All rights reserved.

Keywords: Source code; Linguistic structure; Programming; Comments

1. Introduction

The designers of tools intended to assist programmers often embrace the following reasoning, sometimes implicitly:

- source code has a formal structure defined by the programming language in which it is written;
- in order to be helpful, tools must also understand source code; therefore
- tools should be designed around the formal structure of programming languages.

This reasoning naturally leads designers toward architectures that mimic compilers, the original language-based tools.

Compiler-oriented architectures are inadequate, however, for many kinds of tools, in particular tools that produce transformed source code for ongoing development. These include language translators [1], prettyprinters, automatic restructurers [6], editor auto-indenters, interactive tools for object-oriented refactoring [10], and a broad class of tools for software reengineering. Powerful integrated development environments increasingly support such functionality, sometimes embedded in a source code editor.

A compiler-oriented approach for such tools typically:

1. reads textual source code from files;
2. creates a data structure that represents the formal linguistic meaning of the code, based on some kind of syntax tree;
3. analyzes and/or transforms this data structure;
4. produces a result; and
5. exits, discarding the data structure.

This approach fails for the class of tools mentioned above because it is based too narrowly on formal linguistic[1] structure, and consequently discards nearly all traces of another aspect of code: its documentary structure.

Unlike the textual representation of linguistic structure, which includes keywords, identifiers, operators, and punctuation, *documentary structure* consists of those textual aspects explicitly defined to be *not part of the language*: white space (new lines, spaces, tabs), comments, and choice of names.

Viewed differently, documentary structure is what programmers add to source code for the *sole purpose* of aiding the human reader. This is of enormous importance because of the central role of reading during software development [14]. Programmers clearly understand this: they arrange code carefully, complain about inadequate comments, and argue passionately about the exact placement of braces in code (purely a matter of white space in most languages).

It is almost tautological that documentary structure is *outside* the formal language. It is a much more subtle fact that documentary structure is mostly *orthogonal* to language

E-mail address: michael.vandevanter@sun.com (M.L. Van De Vanter).

[1] In this paper, *linguistic* refers exclusively to programming languages.

0950-5849/02/$ - see front matter © 2002 Sun Microsystems Inc. Published by Elsevier Science B.V. All rights reserved.
PII: S0950-5849(02)00103-9

structure. An important consequence is that compiler-oriented tools do not represent documentary structure adequately. Compilers discard this information freely because it is not needed: humans seldom read compiler output. For other language-based tools, however, losing documentary structure violates the tool builder's equivalent of the physician's oath to 'first do no harm.'

Designers of successful code transformation tools must recognize the following realities:

- text containing source code is a *document* in the human sense of the word;
- a code document is written for both humans and tools, with the human audience being the more important; and
- the documentary structure of code is grounded in information that *cannot be derived from* its linguistic structure, and in fact *cannot even be understood* in those terms.

Builders of language-based tools have long struggled with comments and white space [1,2,6,19,20,29,38]. Variations of language syntax and editing tools have been proposed, but with little success. More recently, JavaDoc comments have some very useful linguistic structure (and a batch-oriented tool set to match [11]), but do not eliminate the need for conventional comments and use of white space. Legasys, a successful reengineering system, honors documentary structure by implementing all code changes as local, carefully computed text changes to the original text [7,24].

The first two realities mentioned above have been recognized: source code is a document whose human legibility is paramount. This paper emphasizes the third with an examination of documentary structure and its consequences for the design of language-based tools. Section 2 begins with background on documentary structure and on how simplistic tools attempt to deal with it. The expedients adopted by such tools, which never seem to work quite right, fail for reasons described in more detail in Section 3: the orthogonality of documentary and linguistic structure. Section 4 describes relevant characteristics of documentary structure and discusses its relationship to frameworks for program understanding, in which it has largely been ignored. Section 5 discusses architectural strategies for preserving documentary structure: and describes why it is so difficult to apply compiler-oriented approaches effectively. Section 6 reviews other approaches that have been taken in dealing with the 'comment problem' and argues that they are not likely to eliminate current mechanisms any time soon. Section 7 concludes with observations, implementation status, and open questions.

2. Background

This section describes in more detail the context of the issue: language definitions, simplistic structure-based transformation tools, the fundamental mismatch, and the apparent technology bias that makes the disconnect so hard to see.

2.1. Programming languages

The documentary structure of source code is dominated by the spatial arrangement of program elements and comments as they appear to a reader on a printed or virtual page. Programmers create this structure using white space and comments, the only tools at hand.

The C++ and Java™ programming languages are typical, with nearly identical treatment of white space and comments. Source code is presumed to be stored in files containing text characters. *White space* is defined to include those characters that are permitted but which do not comprise tokens: space characters, tabs, and line breaks. Tokens are the lexical elements of a program, so by definition white space is *not part of a program*. Furthermore, *comments* are equivalent to white space; they take two forms, *block comments* and *line comments*, as shown in Fig. 1. A complete treatment of the topic occupies two of the 500 pages in The Java Language Specification [15].

A slightly different aspect of documentary structure is the programmer's choice of *names* for computational entities such as classes, methods, and variables. Although an identifier that represents a name is part of the language, and there are some restrictions on which characters can be used, the natural language connotation (i.e., the choice of name) is not; it is properly part of the documentary structure, and is no less important than white space and comments.[2]

For the purpose of this paper, then, the documentary structure of text-based programs consists of these elements:

- Indentation: spaces that separate code or comments from the left margin of the page.
- Inter-token spacing: spaces between adjacent tokens on a line.
- Line breaks: special characters that cause the immediately following character to begin a new line.
- Comments: as shown in Fig. 1.
- The choice of names for language entities.

One cannot help but note the formal weakness of these elements when compared to the rich structure of programming languages. A few languages have offered slightly more structure in white space and comments, but the pre-lexical (i.e., linguistically transparent) approach now dominates.

[2] There are other kinds of structures as well, including the use of the language: idiom, plans, etc. Those are very important (as surveyed by Détienne [9]), but are beyond the scope of this paper.

```
/* the text of a block comment may
   contain line breaks. */

// a line comment ends at a line break
```

Fig. 1. Conventional text comments.

2.2. Programming practice

In contrast to their simple definitions, the use of white space, comments, and names has a long and colorful history, perhaps the more so because formal structure is lacking. Because white space and comments can occur just about anywhere, programmers feel free to create elaborate conventions for their use. Naming conventions are likewise barely restricted.

For example, the code in Fig. 2 (excerpted from a large program written by experienced C++ programmers[3]) nicely demonstrates how programmers make code easy to read. The combination of appropriately terse comments, blank lines, a repeating pattern of layout and suggestive variable names (all are necessary) gives the human reader a tremendous advantage in understanding both the overall point of the code, and the individual clauses that it comprises.

Since reading code is the principal activity of programming, even while writing [14], documentary structure has significant impact on programmer productivity. Programmers know this. For example:

- they demand auto-indenters, whose sole function is to manage white space;
- they argue passionately about the (linguistically insignificant) placement of braces and whether to use them at all when they are optional;
- they debate naming conventions and complain when they are not followed.

Laboratory experiments have shown that improved visual presentation of source code (largely involving documentary structure) increases reading comprehension [3,28,34]. In a backhanded way, Roedy Green makes the same case in his satirical (and well received) essay ‘How to write unmaintainable code’ [16]. Many of his techniques pervert documentary structure in order to obfuscate linguistic structure, clear acknowledgement of the power of the former over the latter.

2.3. Language-based transformation tools

Language-based tools operate on the formal linguistic structure of programs. The conventional data structure for representing programs is a *syntax tree*, derived from source code by *parsing*; some syntactic details may be elided, and additional annotations on tree nodes capture information such as data types. This technology was developed for compilers, the original language-based tools, and we call such approaches *compiler-oriented*.

```
// completely before
if (delend < start) {
  e->start_index += inserted - removed;

// overlap start
} else if (pos <=start && delend < end) {
  e->notify(0, delend - start, 0);
  e->start_index = pos + inserted;

// completely overlaps
} else if (pos <= start && delend >= end) {
  delete e;
  iter.remove_entry();
      .
      .
      .
// completely after
} else {
  // do nothing
}
```

Fig. 2. C++ code with white space and comments.

The tools of most interest in this paper modify programs represented in such an internal representation and then produce source code as a result. In a compiler-oriented architecture this is done by *unparsing*: generating textual source code from a syntax-based representation. For example, language translation systems read programs written in one language and write equivalent programs in another language or a newer version of the same language. Restructuring tools change programs for a variety of reasons, for example the handling of Y2K dates. Tools are currently being explored to support the Extreme Programming [5] practice of ongoing code improvement via object-oriented *refactoring*: reorganizations that do not change the behavior of the program, but which increase the maintainability, and thus the quality of the code [10].

When such a tool is interactive and visual it is called an editor. Compiler-oriented editors are called *structure editors*, or *syntax-directed editors*. Because they represent programs only as syntax trees, they use unparsing to produce a textual display for humans.

2.4. Structural mismatch

The defining characteristic of such tools is that they produce source code for use by people, so maintaining documentary structure is essential. However, because white space and comments exist outside formal program structure, they have no well-defined representation in syntax-based data structures. Consequently, tool designers using compiler-oriented architectures must invent ad hoc strategies for attaching comments and formatting information to syntax trees. The results have generally been unsatisfactory:

[3] All examples are excerpted from professionally written code and have been adapted slightly for compactness.

- A COBOL restructuring system was observed to produce ‘dangerous’ and ‘misleading’ comments because the system was unable to determine which syntactic structures comments originally described [6].
- Language-based structure editors, such as the Program Synthesizer [33], permitted comments only in certain places and gave programmers very little control over their layout. This, along with other restrictions on text-oriented editing, contributed to the perceived inflexibility of such editors, a significant obstacle to their adoption [25].
- JavaML, a proposed standard structural representation for programs written in the Java programming language, stores comments (which the author found ‘especially troublesome’) as attributes on ‘certain important elements [tree nodes]…’ The author notes further that “Determining which comments to attach to which elements is challenging; the current implementation simply queues up comments and includes all that appear since the last ‘important’ element in the comment attribute of the current such element” [2]. This amounts to no real strategy at all.
- A Pascal-to-Ada translation system retained comments by attaching them to tree nodes using simple rules, but the authors admitted that comments would not end up in the same place [1].

In all such cases documentary structure is lost. The cited consequences might be acceptable in the context of infrequently performed tasks, during which there might be careful human review and correction, but they are otherwise unacceptable.

Addressing the “comment problem” in systems such as these is often an afterthought and usually begins with an ill-considered strategy of the sort mentioned above: attach each comment to the ‘right’ place in a syntax tree. Unsatisfactory results are often blamed on not getting the rules right: the rules for attaching comments to tree nodes, and the rules for unparsing them.

In fact, the rules will never be right. The syntax-based strategy, intuitively appealing to language technologists, fails for a fundamental reason: the documentary structure of source code is largely orthogonal to its linguistic structure. Consequently, any naive projection of documentary structure onto linguistic structure necessarily loses crucial information, without which no unparser can produce undamaged source code.

Furthermore, even attaching comments appropriately would not be enough. A considerable amount of the richness in documentary structure, as Sections 3 and 4 point out, derives from white space and its relationship to comments and language elements.

2.5. Technology bias

A surprising aspect of this observation has been how difficult it is for many tool builders to accept. Some kind of bias keeps language technologists from appreciating the huge gap that separates the way languages are formally defined from the way they are used in practice. People understand programs in ways that have relatively little to do with grammars [9], and ignoring this reality is a failure of user-centered [26] design.

The genesis of this paper was a 1993 argument over an experimental tree-based programming environment. Every objection to the proposed strategy, attaching comments to syntax tree nodes, was heard as an admission of failure, i.e., inability to discover the right rules. A subsequent white paper that cited examples from the participants’ own code increased the emotional intensity but failed to modify already hardened positions.

Numerous conversations since have replayed the scenario, most recently before an audience of experts in program analysis and reengineering. As always, the first reaction is to deny that the problem exists—implicitly assuming that documentary structure (if one cared) could be mapped satisfactorily onto syntax trees (if one tried hard enough). Once the case is made, which invariably requires examples of the sort reported in Section 3, the reaction turns to irritation (at being bothered with something as uninteresting as comments) and anger (at programmers who write ‘stupid’ comments). Subsequent, more thoughtful conversation invariably leads to strategies for persuading programmers to change their ways.

The sole exception in the author’s personal experience was the successful Legasys system, whose designers recognized the problem and made solving it a fundamental requirement [7,24].

3. Documentary vs. linguistic structure

This section demonstrates, largely through examples, the fundamental orthogonality between documentary and linguistic structure that was identified in Section 2.4. A consequence is that documentary structure cannot be expressed or even understood in terms of linguistic structure. Examples will show the following:

- The notion of a single comment is itself ill-defined.
- Some white space, in particular line breaks, can be as important as comments.
- The structural referent of a comment cannot be reliably inferred, might not be explicitly represented, and may not exist at all.
- The meaning of textual comments often depends on white space and other comments in ways that defy linguistic analysis.

3.1. An introductory example

Fig. 2 is instructive. The experienced C++ programmer quickly recognizes a sequence of conditional clauses, nicely articulated by indentation and intervening blank lines, each prefaced by a terse comment. A glance at the parallel prose in the comments (‘completely before’, ‘completely overlaps’, etc.), together with the variable names appearing in the Boolean conditionals (‘start’, ‘end’, etc.), make clear that the clauses pertain to possible ordering relationships. Further examination of the code confirms that this interpretation is what the author intended.

Note the location of the second comment, however: it sits *inside* the code handling the first case and thus has no syntactic relationship at all with the code to which it obviously refers.

The first comment is also curious. It precedes the single nested conditional statement that comprises the entire code excerpt, and so might be thought to refer syntactically to the whole thing. The parallel positioning of the nearby comments, however, combined with parallel language in their texts, suggests that it applies only to the first ‘if’ clause.

The final comment is more curious yet: it apparently applies to no statements at all (it is in an empty block) and to no explicit case (there is no expressed Boolean conditional). Many compilers would discard not only the comments, but also the entire ‘else’ clause and its empty block, even though they collectively convey crucial information to human readers.

This discussion is not meant to argue for a particular style of writing comments; many programmers would have commented the code in Fig. 2 differently. The important points are:

- the code is intelligible to humans;
- much of the initial information ascertained by the reader comes from its documentary structure in which even line breaks participate significantly;
- many elements of documentary structure carry meaning only in the context of the whole; and
- the relationship between these elements and the formal linguistic structure of the programs is idiosyncratic at best.

The remainder of Section 3 discusses in more detail these relationships and the disconnect between the documentary and linguistic structures, starting with the most basic issue.

```
storage_size
StructRegion::get_region_size() const
{
 if (size==0){ // ARM(p.164): empty classes
 return 1;     // have nonzero size.
 }
 return size;
}
```

Fig. 3. One comment or two?

```
// This is an extended comment.
// Comments can be very long and might
// extend for several paragraphs.
```

(a)

```
// This is an extended comment.
   // Comments can be very long and might
   // extend for several paragraphs.
```

(b)

```
// This is an extended comment.
//    Comments can be very long and might
// extend for several paragraphs.
```

(c)

Fig. 4. Extended comments.

3.2. Identifying comment boundaries

Any attempt to treat comments formally encounters the immediate problem that comment boundaries are not well defined. This makes it impossible from the outset to think about comments as elements of linguistic structure.

For example, does the method in Fig. 3 contain one comment or two? To the human reader there is only one, but according to the language definition there are two. Treating these comments separately loses information.

Other common configurations exhibit related problems. The code in Fig. 4a contains three comments linguistically, but only a single comment to the human reader. What if the second and third were indented differently than the first (Fig. 4b)? Alternatively, what if the *text* of the second comment were indented several extra spaces, as if at the beginning of a paragraph (Fig. 4c)?

Should an empty comment define a boundary between two adjacent comments, as in Fig. 5, or should it be treated as a paragraph break in a single comment? What if these were block comments instead of line comments, or if they were indented differently? None of these questions have good answers.

3.3. White space as comments

Although comments are widely understood to act as white space, the converse is seldom appreciated: often white space acts as a comment. The most common example is the use of blank lines to group lines of code for the benefit of the reader, sometimes with adjacent comments. Examples of grouping include:

- variable declarations (related in the author’s mind);
- groups of statements (likewise related); and
- statements with associated comments.

For example, a programmer grouped statements in Fig. 6

```
// Finished with that.
//
// Now start this.
```

Fig. 5. One, two, or three comments?

```
...<method header>    {

// Store the default fields
s.defaultWriteObject();

// Store the arrayTable values:
Object[] keys = getKeys();
int validCount = 0;

<etc.>
```

Fig. 6. Statement groups.

using blank lines and added a comment applicable to each group. Even if the comments were retained in a structural representation, and even if they were unparsed back into the same sequence, information would be lost if the blank lines were not reproduced. For example the first comment might be thought to refer to the whole block, and the second might just as well refer backwards.

The documentary strength of blank lines cannot be overstated. Often, blank lines dominate syntax in the mind of the reader. For example, blank lines in Fig. 2 effectively preempt syntactic structure. Without the preceding blank line (and the absence of a following blank line) the second comment in Fig. 2 would be read as referring to the first clause of the conditional statement instead of the second.

3.4. Finding structural referents

Attaching a comment usefully to a syntax tree is often assumed to mean finding the right node: the one to which the comment refers. Even in cases where a right node exists, identifying that node requires understanding the documentary structure.

For example, to a human reader the first comment in Fig. 7 clearly refers to the argument ‘proc_body’ because the two are on the same line. From the linguistic perspective, however, the comma that separates ‘proc_body’ from the comment creates a more natural (linguistically closer) association with the argument ‘static_link’ on the following line.

Fig. 8 makes these relationships more explicit by repeating the example of Fig. 7, somewhat abbreviated, in two forms. The human reader associates the comment backward in Fig. 8a and forward in Fig. 8b.

The expression in Fig. 9 (excerpted from the argument of a return statement) exhibits similar behavior. Three comments contain information crucial to understanding comparisons of bit sequences. The third comment sits completely outside the return statement syntactically, to the right of the terminating ‘;’, but this comment actually refers backward to one of the most deeply nested nodes in the syntax of the preceding statement.

```
push_frame(ic,
  frame_size,
  proc_body,           // frame's “code”
  static_link,         // frame's static link
  ic.get_frame());     // frame's index
```

Fig. 7. Documentary reference in a sequence.

```
pf(ic,
  fs,
  pb,         /*1*/
  sl,         /*2*/
  i.g()) ;    /*3*/
```

(a) with line breaks

```
pf(ic, fs, pb, /*1*/ sl, /*2*/ i.g());/*3*/
```

(b) without line breaks

Fig. 8. Fig. 7 abstracted.

Finally, referring again back to Fig. 2, the second comment refers to code in a different clause of the conditional statement than the one in which it appears. In this situation, documentary structure (grouping and blank lines) causes the comment to refer forward across major syntactic boundaries: past braces, past an ‘else’ keyword, to a different major code block.

It is tempting to consider such cases idiomatic, amenable to recognition by heuristic rules. Even that is doomed to fail when the actual reference depends on the natural language content of the comment(s). Referring once again back to Fig. 2, the parallel use of natural language in sibling comments resolves ambiguity. This problem arises even in simple sequences such as in Fig. 10.

3.5. Missing structural referents

Section 3.4 demonstrated how the true structural referent of a comment can be difficult or impossible to infer. In some cases it may not exist at all.

For example, the final comment in Fig. 2 refers to an implicit Boolean conditional, which can only be understood in the context of all preceding conditionals (actually, even the notion of ‘preceding’ is misleading, since the conditional statements are syntactically nested). The final comment actually refers to the *absence* of any statements.

A language purist might object that the final comment in Fig. 2 actually refers to an invisible ‘empty statement list’. No such objection is likely in Fig. 11. The comment (or is it two?) refers to a method, defined in a separate interface that is not explicitly mentioned at all in the immediate code.

Sometimes the referent is present, but not represented explicitly in conventional data structures. For example, the second comment in Fig. 6 clearly refers to the following *pair* of statements, for which there is no natural representation in a typical syntax tree. Many tree representations for statement sequences are possible, but in none of them would there be a node corresponding precisely to those two statements.

```
(a==b ?  0 :   // Values are equal
 (a<b ? -1 :   // (-0.0, 0.0) or (!NaN, NaN)
  1));         // (0.0, -0.0) or (NaN, !NaN)
```

Fig. 9. Documentary reference in an expression.

```
statement1;
// comment
statement2;
```

Fig. 10. Which statement is the referent?

Finally, there are comments in statement sequences that refer only to the *place* between successive statements, for example as in Fig. 5, to note how much progress toward some goal has been made at this point in the sequence.

3.6. Control flow and stylistic variation

Finally, the interaction between comment placement and control flow can be extremely nuanced, with human interpretation influenced by apparently unbounded variation in the relative placement of comments, line breaks, and braces. This presents challenges to tools that adjust what would otherwise be considered *stylistic* options, for example the use and placement of optional braces.

For example, Fig. 12 shows four presentations of ‘`if`’, the most elemental control statement, along with typical comment locations. These presentations by no means exhaust the possibilities permitted by most languages. The `else` (Fig. 2), for, and `switch` constructs introduce their own complexities.

As discussed previously, some comments may refer to syntactic elements and some may not. For example ‘`c2`’ and ‘`c10`’ may refer to the Boolean condition, the action, or neither. ‘`c5`’ might also refer to the condition, although it is syntactically distant because of the intervening brace.

Multiple comments, possibly with different referents, are often syntactically indistinguishable. For example, ‘`c2`’ and ‘`c3`’ are in the same place from a linguistic perspective, as are ‘`c7`’ and ‘`c8`’. Indentation plays a significant role in these relationships. For example, comments ‘`c5`’, ‘`c6`’ (if present) and ‘`c7`’ might be related and might even be parts of the same documentary comment (as in Fig. 3).

A code transformation tool that attempts to convert the style of an ‘`if`’ statement from one of the presentations in Fig. 12 to another must make difficult decisions in the presence of comments.

4. The documentary structure of code

Section 3 demonstrated that the documentary structure of source code is not related to the linguistic structure in any tractable way. In order to avoid damaging source code, then, language-based tools of the sort under consideration in this paper must be designed in ways that go beyond simple compiler-oriented approaches. This section discusses the evidence concerning documentary structure, evidence that is necessary for understanding architectural goals and trade-offs.

```
public abstract class C implements P {

   // Force this to be implemented
   // public Object anInheritedMethod()

   <etc.>
}
```

Fig. 11. Phantom referent.

```
if (b) statement;  //c1

if (b)    //c2
   //c3
   statement;  //c4

if (b)  { //c5
   //c6
   statement;  //c7
   //c8
   }  //c9

if (b)    //c10
   {  //c11
   //c12
   statement;  //c13
   //c14
   }  //c15
```

Fig. 12. Comment positions for ‘`if`’ statements.

Section 4.1 reviews the research literature on the topic, of which there is surprisingly little, perhaps for reasons similar to those mentioned in Section 2.5. Programming practice, however, is rich in this area, and it informs the taxonomy and analysis of Section 4.2. The remainder of the section describes characteristics of documentary structure that have important implications for tool design:

- it is primarily visual;
- it uses natural language;
- relationships matter; and
- it is robust when compared with linguistic structure.

4.1. Documentary structure in the literature

As crucial as documentary structure is to the human reader, it has been surprisingly neglected in the research literature on programmer psychology. Détienne's comprehensive survey Software Design—Cognitive Aspects includes two chapters on ‘software understanding’ that barely mention it [9]. For example, a proposed two-level cognitive model for program understanding rests on a ‘microstructure’ whose definition mentions only formal syntactic elements.

In contrast, studies have shown that high-quality, fine-grained typography (which is certainly more ‘micro’ than syntactic elements) can contribute significantly to the ability of programmers to understand programs on paper [3,28].

Détienne cites an experiment that asked readers to infer statement grouping as a test of comprehension, but nowhere is it mentioned that an author might use blank lines to communicate this information directly. Détienne frames program understanding in terms of multiple 'schemas', chunks of knowledge in various frames of reference (e.g., elementary, algorithmic, and implementation), but with no mention of authorship. Détienne does recognize that there are 'other' types of schema, and documentary structure is certainly one of them. However, documentary structure differs from others by virtue of its explicit construction by programmers.

The most relevant body of cognitive research comes from Green, who has explored many of the implications of program notation. Green's 'cognitive dimensions' framework addresses how people understand technical artifacts [17]. Among the dozen or so dimensions is 'secondary notation', of which documentary structure is an example. Green and Petre note that this dimension has been 'little studied' for programming, even though it is considered 'indispensable' in other domains, and they see the absence of sufficient secondary notation mechanisms as defects in other design environments [18].

One finds information related to documentary structure mainly in the literature on programming practice and tools. For example, published 'coding conventions' and 'style guides' offer sometimes quite detailed recommendations about the appropriate and uniform use of documentary structure [12,32]. Recommendations even appear in language reference manuals, for example a five-page discussion of naming conventions in The Java Language Specification [15]. Popular text editors such as Xemacs include customizable editing support for particular styles [39].

A magnificent example from the practical literature is Roedy Green's popular satirical essay, mentioned earlier, on writing 'unmaintainable code.' The essay has evolved into a growing on-line collection of 'techniques' for thwarting the human reader [16], clearly motivated by painful experience with incomprehensible code. The discussions accompanying Green's techniques implicitly testify to the importance of documentary structure and will be cited throughout this section.

4.2. The elements of documentary structure

Programmers create documentary structure using the elements listed in Section 2.1. Although not part of any formal framework, each element has its own customs, folklore, and tool support. This section revisits those elements from the perspective of tool design. Questions to be asked about each element include:

- How much documentary information does the element carry?
- How much of the documentary information is redundant, i.e., can be reconstructed from other information (as opposed to irrecoverable information dealing with the author's intent)?
- What level of collaboration between programmers and their tools (typically editors) is customary for managing the elements?

The answers vary significantly, with important implications for the design of language-based tools.

4.2.1. Indentation

No program editor is considered complete without automatic indentation, and the degree to which programmers rely upon it is revealing. By nearly universal agreement, indentation follows syntactic structure, but to the human reader it works the other way: syntactic structure is inferred from indentation, without which programs are incomprehensible.

Although indentation is theoretically redundant, since it can easily be computed from syntax, there are two serious problems with this view.

The first problem is that programs being edited are seldom syntactically perfect. From a formal linguistic point of view, ill-formed programs have no defined syntactic structure at all, meaning that a syntactically driven indentation engine would almost never be of any use. To a programmer, of course, this point of view is preposterous: a syntax error is really just a temporary anomaly [35], and the problem is that syntactically driven tools seldom understand what the programmer is doing. Practical indentation engines are rather more complex:

- they only analyze a tiny subset of language structure (a sort of 'fuzzy parsing'), so that the vast majority of syntactic imperfections are not seen;
- they operate locally in many situations, for example indenting a single line relative to the previous line; and
- they rely heavily on history, leaving most indentation unchanged most of the time, so the programmer can judge when to attempt global reindentation.

In other words, practical indentation in editors relies on syntax as little as possible.

The second serious problem is that indentation is as important to comments as it is to language elements, perhaps more so, but comments have no syntactic structure from which to derive indentation. There are conventional locations for many comments, sometimes discussed in style guides, but many of the examples in Section 3 show comments for which indentation carries crucial documentary information that cannot be computed from anything else.

Indentation is thus redundant, but only for language elements and only when programs are syntactically perfect. The rest of the time, indentation carries important, nonrecoverable documentary information.

Programmers are accustomed to delegating responsibility for indentation completely to tools, although they have conflicting opinions about what those rules should be.

Contention arises when programmers wish to view code using a personal choice of indentation rules (among others), which interacts badly with excessively literal change control systems.

4.2.2. *Inter-token spaces*

On the other hand, what to put between adjacent language tokens within each line is often left to programmer preference. Although it is understood to be important, there are only a few widely established customs such as 'a blank space should appear after commas in argument lists' [32] and 'spaces may not be used between procedure names and their argument list' [12]. Other rules ensure that keywords and parentheses are separated with spaces, as for example in '`while (true) {`', precisely to distinguish them from procedure calls in which the space is discouraged.

Programmers sometimes fight over other details of inter-token spacing, but more as a matter of legibility than any recording of programmer intent. A significant problem is that the space character is too coarse-grained for all situations.

Graphical program designs developed by Baecker and Marcus exploit fine grained control over inter-token spacing to aid visual comprehension [3]. The CP source code editor, a research prototype, demonstrates that this level of typography can be computed from style rules in real time while a programmer types [37]. This technology allows programmers to delegate inter-token spacing completely to their tools, as with indentation now.

Ordinary inter-token spacing thus carries only a moderate amount of documentary information but is largely redundant. A significant exception is the use of extra spacing for alignment across multiple lines. For example, the author of the code in Fig. 13 uses space characters to do manually what tabs do automatically in word processors, automation that is sadly lacking in code editors. Over and above the tedium of manual tabbing, the need to do so discourages the use of proportional fonts, which are in many respects easier to read. This use of inter-token spacing carries a fair bit of documentary information, but can be considered at least partially redundant. Stereotypical uses such as this could be largely implemented by syntax-driven rules, as long as the programmer is given appropriate control.

4.2.3. *Line breaks*

Lines breaks affect the *shape* of code more than any other elements and thus carry a huge amount of documentary information. In practice, the majority are redundant: they appear in conventional places, for example between statements and declarations, and could thus be created from the syntax using rules.

```
int     i          =  0;
int     increment  =  1;
double  sum        =  0.0;
```

Fig. 13. Inter-token spacing for alignment.

The exceptions matter, however. Even without comments, the method call in Fig. 7 reads very differently than it would without line breaks in the parameter list. Line breaking attracts controversy in combination with braces and parentheses, as suggested by the '`if`' statement examples in Fig. 12. These choices are often considered a matter of style, but they can affect the interpretation of comments.

Even more significant are blank lines. Some recommended usages are largely redundant, for example between class declarations, but others are not, for example separating declarations and statements into groups reflecting the programmer's intent.

Given the consequences, as well as the ever present controversies, source code editors often defer to programmers in the placement of line breaks [39].

4.2.4. *Comments*

Ordinary comments carry two kinds of documentary information: content and placement. As examples in Section 3 showed, both affect how the reader understands code.

Placement is managed using white space: line breaks and spaces playing the role of tabs, as mentioned above. Some editors provide support for placing comments in conventional positions, but none of this is recoverable from any other information. Multi-line comments, as discussed in Section 3.2, are especially difficult to recognize when composed of many 'line comments.'

Content is by definition irrecoverable, and there is typically very little support from editors, seldom more than simple paragraph filling. An important characteristic of comments is that the linguistic structure of their content is entirely disjoint from that of the surrounding program, suggesting that perhaps different editing support would be appropriate as well. The CP prototype source code editor demonstrates that this kind of specialized support is possible by treating comments as if they were embedded documents, written in a different language using a different editor, but viewed seamlessly in place [37].

Two exceptional kinds of comments are worth mentioning, as they have rather different properties than those mentioned above:

- Quasi-syntactic comments are much more constrained. For example JavaDoc comments are restricted to well-defined syntactic positions, and their internal structure is partially subject to formal definition [11].
- Graphical comments are composed for their appearance rather than their content, for example rows of asterisks. These often play the role of graphical elements such as lines and boxes.

4.2.5. *Names*

Although expressed in the linguistic structure of the language, names are chosen solely for documentary value.

A reliable sign of a mature and disciplined programming organization is adherence to rigid naming conventions that help make code readable. The vocabulary of such naming conventions typically has a strong natural language component, for example how and when to use verbs, nouns, and adjectives.

Tool support for the choice of names is rare, with the exception of some code auditors that can be programmed to check names against conventions. An interesting exception is Baker's system for translating Common Lisp programs into Ada; names are formed rather differently in the two languages, and considerable effort, ingenuity, and judgment was required to preserve as much of the documentary value of names as possible [4].

4.3. Documentary structure is primarily visual

White space and comments are artifacts of the *visual* aspect of source code: its appearance on a two-dimensional page, either real or virtual. Programmers take great care with this, working as visual designers in addition to their other design responsibilities.

The arrangement of information on a page profoundly influences how people read it. That is why typography and graphic design are applied to the production of human documents: the more difficult the subject matter, the more important they become.

Even the *shape* of code is important. The examples in Figs. 2, 7, and 9 demonstrate that the human reader, presented with conflicting information about the relationship between comments and code, will favor the visual over the syntactic. In fact, there is evidence that programmers seldom think much at all about programs in terms of their formal linguistic structure [31].

This notion of document shape appears in many related contexts. For example, a study of paper forms used by physicians showed that the important aspect of the forms' visual design is not their regularity or logical structure, but whether their visual appearance makes the important things immediately obvious [27].

Détienne cites numerous studies of program comprehension showing that readers scan for *beacons*: features that enable experienced programmers to make reliable assumptions [9]. Although not mentioned explicitly in those studies, visual presentation is clearly important in making beacons easy to locate.

Conversely, as Roedy Green points out, visual layout can equally produce unmaintainable code [16]. His example 'pack as much as possible into a single line' suggests that too few line breaks make code hard to read. 'Take advantage of the complex tokenising rules in C and Java by removing all spaces' likewise suggests that too few inter-token spaces makes code less comprehensible. 'Nest as deeply as you can,' and write 'code that masquerades as comments and vice versa' suggest that creating misleading shapes on the page also makes code hard to read.

4.4. Documentary structure uses natural language

Textual comments are primarily written in natural language, of course, and many of the examples shown in Section 3 can only be related to the code and to one another by understanding prose. For example, the two comments in Fig. 3 are actually one, whereas similarly arranged comments in Fig. 7 refer to distinct parts of the expression. The comment in Fig. 11 uses natural language to describe a syntactic element that is not present.

The most powerful role played by comments is to record aspects of the programmer's intent that cannot be expressed directly in the code. All of Green's suggestions for using comments to produce unmaintainable code amount to lying about that intent [16].

Natural language also plays an enormous role in the selection of names for programming language elements. The power of names to elucidate (or obfuscate) is nowhere more clear than in Green's essay: 32 techniques are listed for confusing the reader through names, but a significant number of techniques in other categories use names as well. Examples include 'use single letter variable names,' 'misspell them,' use 'misleading names,' capitalize idiosyncratically, use 'abstract names,' reuse names, and use 'similar-sounding similar-looking' names [16].

4.5. Relationships matter

Documentary structure is evident, not only in the individual elements, but in rich relationships among the elements.

For example, indentation of a single line by itself means little, but the indentation of a comment relative to nearby lines can have a great impact on the reader. Likewise, extra spaces within a line often have meaning only in relationship to adjacent lines, as shown in Fig. 13.

It is possible, as mentioned earlier, to think of a comment as a miniature natural language document embedded in code, but there is more to it than that. In Fig. 2 it is the carefully related placement and content of multiple comments that communicate the designer's intent.

An essential role in program naming conventions is to establish and use a vocabulary of concepts relevant to a particular system, concepts that are explained in natural language comments.

The documentary structure one sees in source code is often well considered and elaborate. Programmers' documentary techniques are related to the ones used by Baecker and Marcus in their advanced paper presentations of programs [3], but relatively crude because of the limited tools available. Those techniques, all of which deal with relationships among the parts, include page headers, horizontal rules, alignment of columns, and marginalia.

4.6. Documentary structure is robust

A final observation about documentary structure, one with enormous implications for the design of language-based tools, is that it is *robust*, whereas linguistic structure is *fragile*.

The discussion of indentation in Section 4.2 noted that source code under development is seldom grammatically correct, meaning that its linguistic structure is *undefined* most of the time. This is a severe handicap to any language-based tool using a compiler-oriented architecture in the sense described in Section 2. The failure of language-based structure editors, which sought to preserve linguistic structure at the expense of flexible editing, can be ascribed to architectures incapable of properly accounting for the relationship between formal language and textual representation *as seen by users* [35,36].

Transformation tools of the kind discussed in this paper face the same problem. Their ultimate acceptance may depend on the flexibility they offer in the presence of imperfect code.

Documentary structure, on the other hand, persists and changes only in proportion (and in direct response to) the programmer's actions when using ordinary editors. This adds even more weight to the argument for primacy of documentary structure, which programmers see and manipulate directly, over linguistic structure, which is invisible, seldom of primary concern, and often broken.

5. Architectures for documentary structure

The goal set forth in the introduction is to find effective ways for language-based tools to modify source code intended for human consumption. The issues described here apply equally to batch tools, for example, reengineering systems, and to interactive tools, for example refactoring editors.

The challenge is to preserve documentary structure, insofar as possible, given the task at hand. Success must be judged in user-centric terms: whether programmers find that the benefits of a particular language-based tool are worth the cost of damage to documentary structure.

This section discusses specific behaviors that are needed in practice and describes why traditional compiler technology does not help. Three architectural approaches are then described, each with advantages and disadvantages.

5.1. When documentary structure matters

The first part of any solution is to determine when careful handling of documentary structure matters and when it does not. Consider three cases.

1. If a particular file has not been affected, then the original source should be simply be reused. This guards against damage to documentary structure that would seem unreasonable and disproportionate to many users.
2. On the other hand, large changes, for example those involving code movement, leave documentary structure highly suspect and thus in need of human repair. Just about any method for keeping comments available will suffice in this situation; the emphasis should be on interactive editing tools that manage documentary structure well, for example, permitting convenient repositioning of comments.
3. In between are the challenging cases: where code is changed, but where a programmer would perceive as onerous any need to inspect and repair every instance.

The third case therefore should be the focus for transformation tools: making small modifications well enough that programmers will generally trust the result. The criterion, necessarily subjective, is that programmers not feel unreasonably burdened dealing with unwanted consequences.

5.2. Why compilers do not help

Documentary structure appears simple and natural to people, but compiler-oriented architectures are fundamentally unsuited to manage this structure, having been well engineered for entirely different purposes. For example:

- Comments and white space are typically discarded in the earliest state of a compiler's data flow: between the text stream and the lexical token stream, well before enough context is available to examine them usefully.
- Even if comments and white space are passed into the token stream, this feeds into a parser that is based on a grammar in which they have no meaning.
- There is no place in this simple pipeline model where alignments between adjacent lines (Fig. 13) can easily be discovered, especially when they involve syntactic constructs.
- Although much of the documentary structure concerns relationships between white space and tokens, concrete tokens are often discarded from syntax trees, leaving no coherent place to record such relationships.

Consider the basic notion of a line. Although fundamental to the human reader, lines have no natural presence in this architecture. However, compilers need to know something about lines for error reporting, precisely so that they can communicate effectively with humans. This requirement is usually met with special mechanisms that lie outside the standard architectural model. Such mechanisms do not generalize well and often do not behave particularly well either.

Consequently, any language-based tool that effectively manages documentary structure will be much more than a lightly modified compiler:

- more analysis must be done during code input, so that the right information is captured;
- a more general data structure must represent both the linguistic and documentary structures; and
- code transformations must preserve as much documentary structure as makes sense in the eyes of the user.

5.3. Three architectural approaches

5.3.1. Approach 1: handcrafted text patching

The most conservative approach to implementing language-based code modification requires a handcrafted implementation of each operation, typically along the following lines:

- derive linguistic structure from program text;
- use the linguistic structure to determine what needs to be changed, specialized for the particular semantics of the operation;
- also specialized for the particular operation, translate linguistic changes into textual changes;
- apply the textual changes and adjust any affected documentary structure as little as possible.

The auto-indentation mechanisms in many source code editors are simple examples of this approach.

This is also the approach being taken by an emerging generation of interactive refactoring tools. These tend to follow recipes proposed by Martin Fowler. A popular example is *Extract Method*, whose description reads: “You have a code fragment that can be grouped together. Turn the fragment into a method whose name explains the purpose of the method” [10].

In simple cases this operation proceeds as follows:

1. The programmer selects lines of code to be extracted and requests the transformation.
2. Using language-based analysis, the tool determines which local variables are used by code in the selection and proposes them as arguments to the new method.
3. The user is given an opportunity to name the new method and possibly modify its signature.
4. The tool textually cuts the selected code and pastes it into a space between methods, along with any selected white space and comments.
5. The tool surrounds the pasted code with two newly inserted lines: a leading signature and a closing brace.
6. The tool adjusts indentation of the pasted code as a block, disrupting its original documentary structure as little as possible.
7. The tool synthesizes a new method call, inserts it in place of the extracted code, and reindents locally with no disruption to the surrounding code.

This approach is minimally disruptive to documentary structure by virtue of the tool designer’s careful attention. It requires no specialized linguistic infrastructure, again by virtue of the designers’ handcrafting of each operation’s semantics.

On the other hand, this approach does not generalize well. The set of available operations is small, and the cost of adding them is high. Programmers are likely to have no opportunity to create or adapt transformations for their own purposes, and there appears to be no generalized analysis and transformation engine for use by other tools. Finally, code formatting capability is no greater than what is otherwise offered in the environment.

5.3.2. Approach 2: automated text patching

This approach is epitomized by Legasys LS/2000, a language-based, design-recovery and transformation system. LS/200 was used successfully to remediate Y2K problems in billions of lines of source code [7]. This batch-oriented system operates in several phases, with humans in the loop in the important places: guiding the system toward correct identification of trouble spots (using rules and naming conventions, for example), and reviewing the final results of the transformations.

Nearly every phase of the rather complex LS/2000 process is implemented using TXL: a powerful, description-driven pattern matching and transformation engine for tree-based data. A notable exception to the use of TXL is the *version integration* phase, during which code changes computed during earlier phases of the process are carried out.

In order to avoid loss of documentary structure, all derived information and proposed changes in LS/2000 are expressed relative to the original source [24]. Changes are made to the original source by merging (also referred to as ‘backpatching’) a newly transformed version (produced by analysis and transformation) back into the original on a minimal line-by-line basis. Even ‘one character of needless difference’ is considered unacceptable. For small changes of the sort characteristic of Y2K remediation, standard differencing algorithms operating over token streams were found sufficient for computing minimal changes between *before* and *after* tokens.

Although highly successful for its intended application, it is unclear how far LS/2000 architecture generalizes from the perspective of documentary structure. Token-based differencing reportedly did not work well for larger changes or those requiring code movement, pushing the burden for computing differences back onto a customized implementation of each transformation, as in the refactoring tools mentioned earlier. Furthermore, the cited examples are in COBOL, a language with documentary characteristics much different from newer languages.

5.3.3. Approach 3: automated unparsing

A more aggressive approach abandons backpatching and produces transformed code entirely, via unparsing or prettyprinting, starting from a language-based representation. This approach, embodied in a prototype under development at Sun Microsystems Laboratories,[4] risks greater disruption of documentary structure, but with greater potential benefits for interactive analysis and transformation.

Such a system must capture, in a language-based representation, the right information so that an extended unparser can reconstruct document structure for a somewhat modified piece of code. This succeeds if the resulting code, aside from the changes, means the same thing to a human reader as did the original.

One approach, mentioned in Section 2.4, is to use heuristics for attaching comments to tree nodes. This approach fails for all the reasons explained in Section 3. Too much irrecoverable (i.e., non-redundant) documentary information is lost, and no amount of rule-driven pretty-printing can put it back.

Another approach is to record every aspect of documentary structure, including all white space and every lexical token, as proposed by Wagner [38]. This also fails, but for different reasons. First, it may impose unacceptable storage requirements for large bodies of code (compilers use very abstract trees for just this reason). Second, Wagner's technique records only the elements of documentary structure, not the structure itself. It records none of the relationships that an unparser must *reinterpret* for modified code.

For example, the two comments in Fig. 3, which the human reader understands as one (following the cue of their alignment) would no longer be aligned should the variable '`size`' be renamed to something substantially longer. The literal white space that originally appeared between the '`;`' and the second comment is of no use to an unparser trying to put this right; the important fact, that the two comments were aligned before the change, is lost.

The approach under consideration is driven by the analysis of documentary structure presented in Section 4. Some documentary information is redundant, and can thus be ignored. Other documentary information is not, and must be recorded in some form. In many cases, it is the relationships that must be recorded, for example the relative alignment of comments on successive lines, rather than the particular columns. Here are specific examples of the information that might be captured:

- Where each comment occurred relative to the original token stream, even if some of the tokens are not explicitly represented in the structure (e.g., commas in argument lists).
- The layout of each comment: its horizontal position relative to adjacent code and comments, the number of line breaks preceding it, and the number of line breaks following it.
- The ordering of adjacent comments (between the same two tokens); in these cases intervening line breaks are counted for both comments.
- The relationship between aligned '//' comments: those that appear on successive lines at the same horizontal position.
- Blank lines (two or more line breaks without intervening tokens or comments).
- Any other 'unusual' line breaks not associated with comments.
- Any other unusual white space in lines, along with discovered alignments of the sort appearing in Fig. 13.
- 'Extra' syntax, for example redundant parentheses and empty blocks of the sort appearing in Fig. 2, especially when associated with comments.

Unparsing rules must be amended to account for documentary structure, which must naturally take precedence over linguistic structure. For example, blank lines must be restored, assuming that their surrounding context is unscathed. To first approximation, comments must be placed between the same adjacent tokens (with the complication that braces and parentheses may appear, disappear, and shift around), and additional line breaks inserted to restore original visual relationships. Indentation may vary considerably, but the alignment of related comments and code must be restored. Special unparsing rules can be applied to array initializers, aligning elements into columns for example, as long as the original line breaks were retained so that the overall shape of the data remains approximately the same.

6. Other directions

Although little has changed in this area for years, formatting and comments have long been seen as problematic. They are awkward for both people and tools, and they have never reached the degree of utility that we intuitively believe possible.

This section reviews other schools of thought on how this might be changed. All have merit, but in the near future none are likely to make unnecessary the requirements described in this paper.

6.1. Literate programming

The most ambitious and successful attempt to rethink the relationship between documentary structure and source code is Knuth's Literate Programming [22]. Knuth begins with the premise that code is primarily a document for humans, and he makes this aspect paramount. Programmers write code fragments and embed them in a rich document dominated by prose. Batch tools produce both nicely formatted documents (prose with code embedded) for human consumption and source code (generally unformatted) for compiler consumption. Knuth and others argue

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

that writing in this style produces better code in the first place [22,30].

Despite having a loyal following, Literate Programming has never been widely adopted, and the reasons are unclear. Perhaps the extra layer of tools was perceived as onerous by programmers (or managers). Perhaps it is not easily adapted to object-oriented programming, a different paradigm for factoring code into small pieces.

Literate Programming is fundamentally incompatible with the class of tools under discussion in this paper. Linguistic structure is not available for analysis and transformation without a separate derivation step from the original literate source; this is a serious obstacle to language-based code transformations. This leads back to the question of how languages are designed in the first place.

6.2. Fix the languages

Another school of thought sees the problem as defective language design. In this view, source code would still be stored in text files containing textual comments, but language definitions should be extended to bring comments into the formal linguistic structure.

Kaelbling, noting the difficulty of understanding the referents of simple textual comments, observes that this can be fixed either by language extension or convention [20]. Acknowledging the practical obstacles to changing language grammars, Kaelbling suggests instead that programmers add explicit 'scope markers' to text of comments, and that analyzers could deduce from these markers the structural referents of the comments.

Grogono starts with much the same objections, noting that "software tools can do very little with comments that are equivalent to white space" [19]. He suggests that future languages include a more general syntactic framework that would include other information, for example assertions and pragmas, as well as comments.

The first widely accepted structural comments appear in the Java programming language [15]. 'JavaDoc' comments are specially tagged and intended to appear only at the beginning of public class and member declarations. Standard batch tools extract interface documentation for hyperlinked publication in HTML [11], but ignore conventional comments placed elsewhere.

A practical difficulty with all these approaches is that widespread adoption of new languages is relatively rare and tends not to be driven by comment mechanisms. A more fundamental difficulty is that merely making comments structural may not be enough; much of the documentary information shown in Sections 2 and 3 is not about syntactic structure at all.

Languages occasionally appear in which white space is linguistically significant. Python programmers specify syntactic nesting using indentation, rather than the more common braces [23]. This changes the way responsibility for indentation is shared between programmer and tools, but it has little effect on the role played by indentation for the human reader.

6.3. Fix the programming environments

Yet another school of thought proposes better tool support for programming language comments.

For example, Robillard refutes Kaelbling with the claim that syntactic extensions to existing languages can be used, as long as tools hide the complexity from the users [29]. He proposes that an extended text editor track the syntactic scope ('referent' in the terminology of this paper), but without convincing detail.

Another class of programming environments replaces the textual representation of source code with purely structural storage that is assumed to permit greater richness. For example, structure editors such as the Synthesizer Generator represent programs only as annotated syntax trees [33]. But comments are seen as little more than annotations on nodes. This reduces programmer control over documentary structure without offering anything new in its place.

Both of the above two approaches are based on the assumption, refuted in Section 3, that comments are 'about' particular syntax nodes.

Smalltalk programming takes place in a structured, browser-based environment [13], and it is no coincidence that current approaches to code refactoring originated in that community. Documentary structure does not confound code transformations in Smalltalk to the extent that it does in other languages because documentary structure in Smalltalk is generally expressed in very different, more structural ways. Class comments, class and method categories, and distinguished comments at the head of methods are all managed structurally by the environment. Furthermore, the custom of factoring Smalltalk code into very small methods elevates the role played by naming, a documentary element that is also managed structurally.

An even more provocative approach is *hyperprogramming*: representing programs as fully typed persistent language objects that can be manipulated by specialized editors [21]. There have been proposals to extend the hyperprogramming model with fine-grained hyperlinks to documentation such as requirements, but there is surprisingly little discussion of how to document the code itself [8]. Such systems require very different languages and programming infrastructures than are widely available today.

6.4. Make comments unnecessary

As in Smalltalk, the more widespread trend toward highly factored object-oriented code, combined with intelligent naming of the parts, reduces the need for interspersed comments. Fowler puts it this way: "How do you identify the clumps of code to extract? A good technique is to look for comments. They often signal this kind of semantic distance. A block of code with a comment that tells you what it is doing can be replaced by a method whose name is based on the comment. Even a single line is worth extracting if it needs explanation" [10].

This is Fowler's *Extract Method* transformation, mentioned in Section 5.3. It can be seen as a kind of lateral move in which one kind of documentary structure (comments associated with a group of lines, as described in Section 3.4) is replaced with another kind (method naming) that presumably carries the same information. Section 3, however, showed many other kinds of documentary structure than the one addressed by 'method extraction.' It certainly does not change the need for intelligent white space layout, nor is it likely to replace the kind of general commentary (explanation, background, and motivation) readers appreciate.

Frequent refactoring is a basic tenet of Extreme Programming [5], and there is a natural interest in tools to support the process. These are among the language-based transformation tools addressed by this paper, and they are likewise subject to the analysis presented in Section 5.1. Programmers may appreciate tools that automate refactoring, but will be unhappy if they must investigate and possibly repair every bit of affected source code. Some of the more ambitious transformations will have widespread effects, making the preservation of documentary structure essential to success.

7. Conclusions, status, and outlook

The documentary structure of code (the use of comments, white space, and naming) is far more important to programmers than one might infer from its treatment by designers of programming languages and language-based tools.

The perspective presented here calls new attention to the job of programming. In addition to other responsibilities, programmers should be seen as authors and graphical designers who take responsibility for the human legibility of their source code. They use the limited means available to them, documentary structure, to make important things obvious.

Advanced programming tools, interactive or batch, that perform language-based source code transformations will find little acceptance without attention to this issue. Strategies for dealing with documentary structure in such tools must ultimately be judged by their success: whether programmers find that the need to repair damage done by tools outweighs their advantages.

The analysis presented here, based on concrete examples from production code, points toward a better understanding of documentary structure and how tools might account for it properly. A corollary is that conventional compiler-oriented architectures for language analysis are fundamentally unsuited to capturing documentary structure.

Some of the strategies presented here were implemented in 1993 as part of an internal project at Sun Microsystems Laboratories, but in a system that was never complete enough for evaluation. A new implementation is currently in progress as part of the Jackpot project at Sun Labs, with the expectation that many of these options can be explored.

Acknowledgments

Discussions with members of the Jackpot project at Sun Labs, Tom Ball, James Gosling, and Tim Prinzing contributed to this paper, as have extensive comments and suggestions from Yuval Peduel. Marat Boshernitsan and anonymous reviewers made helpful comments on an earlier version of this paper.

References

[1] P.F. Albrecht, P.E. Garrison, S.L. Graham, R.H. Hyerle, P. Ip, B. Krieg-Brückner, Source-to-source translation: Ada to Pascal and Pascal to Ada, Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, Boston, MA, USA, 9–11 December, SIGPLAN Notices 15 (11) (1980) 183–193.

[2] G.J. Badros, JavaML: a markup language for Java source code, Ninth International World Wide Web Conference Amsterdam (2000).

[3] R.M. Baecker, A. Marcus, Human Factors and Typography for More Readable Programs, Addison-Wesley, Reading, MA, 1990.

[4] H.G. Baker, Strategies for the lossless encoding of strings as Ada identifiers, ACM Ada Letters XIII (5) (1993) 43–47.

[5] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, Reading, MA, 1999.

[6] F.W. Calliss, Problems with automatic restructurers, SIGPLAN Notices 23 (3) (1988) 13–21.

[7] T.R. Dean, J.R. Cordy, K.A. Schneider, A.J. Malton, Using design recovery techniques to transform legacy systems, Proceedings of the ICSM—IEEE International Conference on Software Maintenance, Florence November (2001) 622–631.

[8] A. Dearle, C. Marlin, P. Dart, A hyperlinked persistent software development environment, Proceedings of Hyper-Oz'92: A Workshop on Hypertext Activities in Australia, Adelaide, Australia (1992).

[9] F. Détienne, Software Design—Cognitive Aspects, Springer, Berlin, 2002.

[10] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999.

[11] L. Friendly, The design of distributed hyperlinked programming documentation, in: S. Fraïssé, F. Garzotto, T. Isakowitz, J. Nanard, M. Nanard (Eds.), Hypermedia Design, Proceedings of the International Workshop on Hypermedia Design (IWHD'95) Montpellier, France, Springer, Berlin, 1996, pp. 151–173.

[12] K. Gabryelski, Wildfire C++ Programming Style: With Rationale, Wildfire Communications Inc, 1997, http://www.literateprogramming.com/wildfire.pdf.

[13] A. Goldberg, Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA, 1983.

[14] A. Goldberg, Programmer as reader, IEEE Software 4 (5) (1987) 62–70.

[15] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Second ed., Addison-Wesley, Reading, MA, 2000.

[16] R. Green, How to write unmaintainable code, Java Developers' Journal 2 (6), updated frequently at http://mindprod.com/unmain.html.

[17] T.R.G. Green, Instructions and descriptions: some cognitive aspects

of programming and similar activities, in: V. Di Gesù, S. Levialdi, L. Tarantino (Eds.), Proceedings of the Working Conference on Advanced Visual Interfaces (AVI 2000), ACM Press, New York, 2000, pp. 21–28.

[18] T.R.G. Green, M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework, Journal of Visual Languages and Computing 7 (1996) 131–174.

[19] P. Grogono, Comments, assertions, and pragmas, SIGPLAN Notices 24 (3) (1989) 9–84.

[20] M.J. Kaelbling, Programming languages should NOT have comment statements, SIGPLAN Notices 23 (10) (1988) 59–60.

[21] A.M. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R. Morrison, R.C.H. Connor, Persistent program construction through browsing and user gesture with some typing, in: A. Albano, R. Morrison (Eds.), Proceedings of the Fifth International Workshop on Persistent Object Systems (POS5), San Miniato, Italy, Springer, Berlin, 1992, pp. 86–106.

[22] D.E. Knuth, Literate programming, The Computer Journal 27 (2) (1984) 97–111.

[23] M. Lutz, D. Ascher, Learning Python, O'Reilly & Associates, 1998.

[24] A. Malton, K.A. Schneider, J.R. Cordy, T.R. Dean, D. Cousineau, J. Reynolds, Processing Software Source Text in Automated Design Recovery and Transformation, Proceedings of the Ninth International Workshop on Program Comprehension (IWPC) Toronto, Canada, IEEE Computer Society, 2001, pp. 127–134.

[25] L.R. Neal, Cognition-sensitive design and user modeling for syntax-directed editors, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Toronto, Canada (1987) 99–102.

[26] D. Norman, S. Draper (Eds.), User Centered System Design: New Perspectives on Human–Computer Interaction, Lawrence Erlbaum, London, 1986.

[27] E. Nygren, M. Lind, M. Johnson, B. Sandblad, The art of the obvious, Human Factors in Computing Systems CHI'92 Conference Proceedings, Monterey, CA, USA May (1992) 235–239.

[28] P. Oman, C.R. Cook, Typographic style is more than cosmetic, Communications of the ACM 33 (5) (1990) 506–520.

[29] P.-N. Robillard, Automating comments, SIGPLAN Notices 24 (5) (1989) 66–70.

[30] S. Shum, C. Cook, Using literate programming to teach good programming practices, Proceedings of the 25th SIGCSE Technical Symposium on Computer Science Education, Phoenix, AZ (1994) 66–70.

[31] E. Soloway, K. Ehrlich, Empirical studies of programming knowledge, IEEE Transactions on Software Engineering 10 (1984) 595–609.

[32] Sun Microsystems, Code Conventions for the Java™ Programming Language (http://java.sun.com/docs/codeconv/), 1999.

[33] T. Teitelbaum, T. Reps, The Cornell program synthesizer: a syntax-directed programming environment, Communications of the ACM 24 (9) (1981) 563–573.

[34] T. Tenny, Program readability: procedures versus comments, IEEE Transactions on Software Engineering 14 (9) (1988) 1271–1279.

[35] M.L. Van De Vanter, R.A. Balance, S.L. Graham, Coherent user interfaces for language-based editing systems, International Journal of Man–Machine Studies 37 (4) (1992) 431–466 (reprinted in Structure-Based Editors and Environments, G. Szwillus, L. Neal (Eds.), Academic Press, 1996).

[36] M.L. Van De Vanter, Practical Language-based Editing for Software Engineers, Software Engineering and Human–Computer Interaction: ICSE'94 Workshop on SE-HCI: Joint Research Issues, Sorrento, Italy, Springer, Berlin, 1995, LNCS 896.

[37] M.L. Van De Vanter, M. Boshernitsan, Displaying and editing source code in software engineering environments, Second International Symposium on Constructing Software Engineering Tools (CoSET'2000), 5 June 2000, Limerick Ireland, ICSE 2000 Workshop Proceedings.

[38] T.A. Wagner, Modeling user-provided whitespace and comments, Practical algorithms for incremental software development environments, PhD Dissertation, Report No. UCB/CSD-97-946, University of California, Berkeley, 1997.

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