IEEE International Workshop on Source Code Analysis and Manipulation, November 2001

SML 2001-0392

Preserving the Documentary Structure of Source Code in Language-based Transformation Tools

Michael L. Van De Vanter
Sun Microsystems Laboratories
901 San Antonio Road, UMTV29-112
Palo Alto, CA 94303 USA
Michael.VanDeVanter@Sun.COM

Abstract

Language-based tools necessarily translate textual source code into grammar-based representations. During translation, tools such as compilers and analyzers are generally free to discard comments and white space, which have no impact on the code's formal meaning. Tools that produce transformed source code for human consumption enjoy no such freedom. Comments and white space are crucial to the comprehensibility and maintainability of source code and thus to its ultimate value. However, it is not always practical or desirable for transformation tools to replicate comments and white space in their entirety. An analysis of the documentary *(as opposed to linguistic) structure of source code leads to a practical strategy for preserving its comprehensibility when processed by such tools.*

1. Introduction

Many language-based tools for dealing with source code follow the design of compilers (the original language-based tools). Such tools typically:

1. read textual source code from files;
2. create a data structure that represents the formal linguistic[1] meaning of the code, based on some kind of syntax tree;
3. analyze and/or transform this data structure;
4. produce a result; and
5. exit, discarding the data structure.

This works well for a large class of tools but is fundamentally inadequate in tools that produce transformed code for ongoing development. Such tools include language translators [1], prettyprinters, automatic restructurers [5], and interactive tools for object-oriented refactoring [7]. The compiler approach fails because it discards a crucial aspect of source code (which we call its *documentary* structure) that is largely orthogonal to its formal linguistic structure. A significant portion of documentary structure is expressed via comments and white space,[2] and in many cases preempts linguistic structure in the eyes of human readers.

Designing useful code transformation tools requires a fundamental change of perspective, away from the compiler view of source code. In this alternate perspective, tool builders must acknowledge that:

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

The documentary structure of source code makes it comprehensible to people; incomprehensible code has no long-term value. Source code tools are designed to add value, but if they lose documentary structure they violate the tool builder's equivalent of the physician's oath to "first do no harm."

Although builders of language-based tools have long struggled with comments and white space [1,2,5,11,12,17,23], for decades there has been no significant progress in the way they are managed. Variations of language syntax and editing tools have been proposed, but without apparent success. There are small signs of change, for example the somewhat more structured JavaDoc comments [8], but the focus of this paper is on conventional

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

2. Other parts of the documentary structure include naming and use of programming idiom, both beyond the scope of this paper.

comments and white space in code written in languages such as C, C++, and the Java™ programming language.

The challenge for the class of tools identified here is to construct, along with modified code, a new documentary structure that conveys the same meaning to the human reader as did the original. This reflects a significantly different focus than the approaches taken by other tools.

This paper argues for this new perspective and describes a strategy for implementing such tools. Section 2 begins with background on white space and comments and on how tools have in the past attempted to deal with them. The expedients adopted by those tools, which never seem to work quite right, are shown in Section 3 to be fundamentally flawed by their neglect of documentary structure. Section 4 describes the relevant characteristics of documentary structure, and Section 5 presents an architectural strategy for preserving it. The proposed strategy is as conceptually as simple as the documentary characteristics identified in Section 4, although it is challenging to implement in conventional language-based frameworks. 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 comment 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 problem: the nature of white space and comments, the design of structure-based transformation tools, and the fundamental mismatch between them.

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, but it is not formally part of the program.

The C++ and Java programming languages are typical, with nearly identical treatment of white space and comments. In both cases the presumption is that source code is stored in text files. *White space* (space characters, tabs and line breaks) is defined to include those characters that do not comprise tokens; tokens are the lexical elements of a program, so white space is by definition *not part of a program*. *Comments* are equivalent to white space and can take two forms (*block comments* and *line comments*), as shown in Figure 1. The entire topic is covered in 2 of the 500 pages in *The Java Language Specification* [10].

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

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

Figure 1. Conventional text comments

One can't help but note the weakness of comments and white space, especially when compared to the rich formal structure of programming languages. There have been languages offering slightly more structure in their white space and comments, but the pre-lexical (i.e. linguistically transparent) approach now dominates.

In contrast, the *use* of white space and comments has a long and colorful history, perhaps the more so because formal structure is lacking. They can occur just about anywhere, so programmers feel free to create elaborate conventions for their use. For example, the code in Figure 2 (excerpted from a large program written by experienced C++ programmers[1]) nicely demonstrates how programmers make code easy to read. The combination of appropriately terse comments, blank lines, and a repeating pattern (all three are necessary) gives the human reader a tremendous advantage in understanding both the overall point of the code, as well as the individual clauses that comprise it.

```
// 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();
... <some clauses omitted>


// completely after
} else {
  // do nothing
}
```

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

Since reading code is the principal activity of programming, even while writing it [9], documentary structure has significant impact on programmer productivity. Programmers know this. For example:

1. All code examples in this paper are excerpted from production code, with some renaming and reformatting for compactness.

- they demand auto-indenters, which only manage white space; and
- they argue passionately about the (linguistically insignificant) ordering of braces and line breaks.

Laboratory experiments have shown that improved visual presentation of source code (largely involving documentary structure) increases reading comprehension [3,16,21].

For the purpose of this paper, documentary structure consists of these elements:

- *Indentation*: spaces that separate code or comments from the left margin of the page.
- *Inter-token spaces*: spaces between adjacent tokens on a line.
- *Line break*: a special character that causes the following character to begin a new line.
- *Comments*: as shown in Figure 1.

2.2 Language-based transformation tools

Language-based tools such as compilers and language-based editors operate on the formal linguistic structure of programs. The conventional data structure for representing programs is a *syntax tree*, derived from source code by context-free *parsing;* some syntactic details may be elided, and additional annotations on tree nodes capture such context-sensitive information as data types.

The tools under consideration in this paper modify programs represented in such an internal representation and then produce a result by *unparsing*: generating textual source code from the internal 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 [4] practice of ongoing code improvement via object-oriented *refactoring*: rearrangements that do not change the behavior of the program, but which increase the maintainability, and thus the quality of the code [7].

2.3 The structural mismatch

The defining characteristic of such tools is that they must generate transformed source code suitable for further use by people, to whom documentary structure is essential. But white space and comments are not formally part of programs, so they have no well-defined representation in these conventional data structures. This leaves tool designers with ad hoc strategies for attaching comments to syntax trees, with generally unsatisfactory results:

- A COBOL restructuring system was observed to produce “dangerous” and “misleading” comments [5].
- JavaML, a proposed standard structural representation for programs written in the Java programming language, stores comments (which are called “especially troublesome”) as attributes on “certain ‘important’ elements [tree nodes] (including class, anonymous-class, interface, method, field, block, loop). ... 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].
- A Pascal-to-Ada translation system retained comments by attaching them to tree nodes using simple rules, but the authors admitted that comments wouldn’t end up in the same place [1].

In all these cases the documentary structure of code is mostly lost. This might be acceptable in the context of infrequently performed tasks, during which humans review and correct all the results, but it fails completely in any environment where transformations occur frequently.

Damage occurs because of the structural mismatch between the documentary and linguistic structures, and because of the belief that attaching comments to their “natural” locations in a syntax tree is sufficient. The following section shows why this strategy, often adopted instinctively, is doomed.

3. Documentary structure is not linguistic

Addressing the “comment problem” in language-based tools is often an afterthought and usually follows an ill-considered strategy of the sort mentioned above: attach each comment to the “right” place in the tree. Although intuitively appealing to language technologists, this fails for a fundamental reason. The documentary structure of source code is largely orthogonal to its linguistic structure. Projecting documentary structure onto linguistic structure loses crucial information, without which no unparser can produce intelligible source code.

In practice, unsatisfactory results are often blamed on not getting the rules right: rules for attaching comments to tree nodes, and rules for unparsing them. This section demonstrates why such rules will never be right, no matter how much language technology is applied. For example:

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

3.1 An instructive example

Referring back to Figure 2, consider the meaning of the second comment: "`overlap start`". The C++ programmer quickly ascertains, even before deciphering the text, that it identifies the second in a list of cases being handled separately. Indentation and blank lines encourage understanding the code as a list of cases with responses, rather than the deeply nested conditional statement which actually implements it.

Once this relationship among line groupings is understood, the text of the comment can then be read in the context of its sibling comments ("`completely before`", "`completely overlaps`", etc.). At this point, possibly with a glance at the boolean conditionals, it becomes clear that the clauses pertain to possible ordering relationships.

The binding between comments and cases is made clear by the juxtaposition of each comment with its associated code, and by separating the groups with blank lines. Note the position of the second comment, however: it sits *inside* the code handling the first case and 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 (it is in an empty block) and to no explicit case (there is no expressed boolean conditional). Many compilers would not only discard 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 Figure 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 these relationships in more detail, starting with the most basic problem in managing comments.

3.2 Identifying comment boundaries

Any attempt to capture comments in source code immediately encounters the problem that comment boundaries are not well defined.

For example, does the method in Figure 3 contain one comment or two? To the human reader there is only one,

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

Figure 3. One comment or two?

but there are two according to the language definition. Treating these comments separately amounts to loss of information.

Other common configurations exhibit related problems. The code in Figure 4 contains three comments linguistically, but only a single comment to the human reader. What if the second comment were indented differently than the other two? Alternately, what if the *text* of the second comment were indented several extra spaces, as if at the beginning of a paragraph?

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

Figure 4. Extended comments

Should an empty comment define a boundary between two adjacent comments, as in Figure 5? What if these were block comments instead of line comments, or if they were indented differently? None of these questions have good answers.

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

Figure 5. One, two, or three comments?

3.3 White space as comments

Although comments are widely understood to act as white space, the converse is seldom appreciated, namely that white space often acts as a comment.

For example, the statements in Figure 6 are grouped by blank lines to show which comments apply to which lines. 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.

```
...<method header>    {

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

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

<etc.>
```

Figure 6. Statement groups

The documentary strength of blank lines cannot be overstated. In Figure 2 blank lines effectively preempt the syntactic structure of the code. Without the preceding blank line (and the absence of a following blank line) the second comment in Figure 2 would be seen in the first clause of the conditional statement rather than 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. However, this can depend crucially on documentary structure.

The second comment in Figure 2, for example, refers to code in a different clause of the conditional statement than the clause in which it appears; in that case, documentary structure causes the comment to refer forward: across braces and across an "`else`" keyword.

Documentary references can also point backwards. In Figure 7 the first comment refers backwards to the argument "proc_body", even though a preceding comma makes the comment linguistically closer to the following argument "`static_link`". Without the line breaks, the first comment would be understood to refer forward, as shown in Figure 8.

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

Figure 7. Documentary reference in a sequence

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

Figure 8. Figure 7 without line breaks

The expression in Figure 9 (excerpted from the argument of a `return` statement) exhibits similar behavior. Three comments contain information crucial to understanding bit sequence comparisons. The third sits completely outside the `return` statement, to the right of the terminating ";", but it actually refers backward to one of the most deeply nested nodes in the preceding statement.

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

Figure 9. Documentary reference in an expression

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), for example in Figure 2 (where the parallel language of sibling comments resolves ambiguity), and in simple sequences such as in Figure 10.

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

Figure 10. Which statement is the referent?

3.5 Missing structural referents

The previous section 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 Figure 2 refers to an implicit boolean conditional, which can only be understood in the context of all preceding conditionals. The comment also refers to the *absence* of any statements, although a language purist might object that it refers to an invisible "empty statement list".

The comment in Figure 11, or more accurately the two comments, refer to a method, defined in a separate interface, that is not explicitly mentioned at all in the immediate code

```
public abstract class C implements P {

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

    <etc.>
}
```

Figure 11. Phantom referent

The second comment in Figure 6 clearly refers to the following pair of statements, for which there is no natural representation in a typical syntax tree. A concrete parse tree might represent sequences as right recursive binary

trees, where each node refers to a single member and to the remainder of the sequence; a more abstract syntax tree might have a single parent for all nodes in the sequence. In neither case, however, is there a node corresponding precisely to those two statements.

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

4. The documentary structure of code

Section 3 showed how the documentary structure of source code is not related to linguistic structure in any tractable way. This section describes documentary structure on its own terms. The good news is that, unlike the complex ways in which the documentary structure of code is *not* related to its formal linguistic structure, the fundamental nature of documentary structure is easily described and has a great deal in common with other kinds of document layout. The examples of Section 3 demonstrated most of it, and this section summarizes its essential characteristics: it is visual, it has several elements, and it mainly depends on spatial relationships and comments.

4.1 Documentary structure is visual

White space and comments are artifacts of the *visual* aspect of source code: its appearance on the two dimensional page (real or virtual) on which humans read it.

The appearance and arrangement of information on a page profoundly influences how people read it. That's 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 Figures 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 [19].

This notion of document shape appears in many related contexts. For example, a study of hardcopy 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 [15].

This perspective casts new light on the job of programming. In addition to other responsibilities, programmers act as graphical designers and 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.

4.2 The elements of documentary structure

Each aspect of documentary structure, following the taxonomy presented in Section 2.1 has its own customs, folklore, and tool support. They differ also in how much information about documentary structure they carry, i.e. information that can not otherwise be computed from the linguistic structure.

Indentation provides essential feedback on nesting structure, which is otherwise difficult to see. It is the best supported, and programmers usually delegate responsibility completely to tools, which compute it from the linguistic structure. As a consequence, indention by itself usually carries little additional documentary information.

*Inter-token space*s, on the other hand, are usually left to programmer preference, since there are few widely established customs. Graphical program designs developed by Baecker and Marcus exploit fine grained control over inter-token spacing to aid visual comprehension [3], and the CP source code editor demonstrates that this level of typography can be computed from style rules in real time while a programmer types [22]. This means that inter-token spacing also carries little additional documentary information. A significant exception is the use of extra spacing for vertical alignments of the sort appearing in Figure 12

```
int i         = 0;
int increment = 1;
```

Figure 12. Inter-token spacing for vertical alignment

Line breaks affect the shape of code the most and thereby attract the most controversy, especially in combination with braces and parentheses. Source code editors usually defer to programmers in the placement of line breaks [24]. Examples in Section 3 show how line breaks carry a great deal of documentary information, both in isolation and in relationship to other elements.

Comments, which by definition carry only documentary information, often receive rudimentary support from editors, for example paragraph filling. The CP source code editor carries this support much further. It treats comments as subdocuments: following a compound editing model, and it instantiates for each comment a sub-editor that is specialized for natural language content [22].

4.3 Relationships matter

Documentary structure emerges from what programmers do with these four elements and in the rich relationships among them.

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 meaning of the comment. Likewise, extra spaces within a line often have meaning only in relationship to adjacent lines, as shown in Figure 12.

Numerous examples in Section 3 showed how the meaning of comments is heavily influenced by page layout (indentation and line breaks) and by other comments whose relationships may be both visual (spatial alignment) and textual (parallel prose).

The documentary structure one sees in source code is often well considered and elaborate. Programmers' documentary techniques can be viewed as crude versions of the ones used by Baecker and Marcus in their advanced paper presentations of programs [3]. Those techniques, all of which deal with relationships among the parts, include page headers, horizontal rules, vertical alignment of columns, and marginalia.

4.4 Documentary structure is robust

It is worth noting in passing that documentary structure is also *robust* relative to the *fragile* linguistic structure of source code. During ordinary textual editing, source code is only occasionally compliant with a formal grammar. Consequently, linguistic structure is *undefined* most of the time, a severe disadvantage for language-based tools in any editing environment.

Documentary structure, on the other hand, persists and changes only in proportion (and in direct response to) the programmer's actions. This adds even more weight to the argument for primacy of documentary structure, which programmers see and manipulate directly, over the linguistic structure, which is invisible, not of primary concern, and often broken.

5. Preserving documentary structure

The challenge set forth in the introduction is to find a way for language-based tools to capture and reconstruct through unparsing enough of the documentary structure of source code that the result is not perceived as unacceptably damaged. This section discusses what this means in practice, mentions some approaches that don't work, and proposes a framework that will.

5.1 When documentary structure matters

The first part of a solution is to determine when the documentary structure matters, and when it does not. Consider the following three cases:

1. If a particular piece of source code hasn't been changed at all by the tool, then no recording of documentary structure is necessary; a simple pointer back to the original source file suffices.
2. On the other hand, dramatic code modifications leave any captured documentary structure highly suspect and in need of repair by a programmer anyway. Just about any method for keeping comments available will be good enough in this situation.
3. In between are the cases where it matters, where the code has been changed somewhat, but not so much that a programmer would accept the need to repair it.

The third case therefore should be the focus: unparsing a somewhat modified piece of code into something "close enough" to the original that a programmer will not feel compelled to review every aspect of the result. This criterion for success is somewhat subjective, which calls for a flexible and general strategy.

5.2 Map comments onto syntax

Attempting to attach comments meaningfully to a syntax tree is the naive approach taken by many tools, such as those mentioned in Section 2.3. From the perspective of this paper, that approach discards too much documentary structure. As demonstrated in Section 3, this is information crucial to understanding the meaning of comments and code, and no amount of unparsing technology can put it back. More information must be recorded when text is translated into a language-based internal representation.

5.3 Keep everything

At the opposite extreme, a tool might record all of the white space found in the original source text, and by implication all of the lexical tokens of the program [23]. This approach has two shortcomings. First, it may impose unacceptable storage requirements for large bodies of code.

More seriously, this strategy is insufficient; it records only the elements of documentary structure, not the structure itself. It records none of the relationships that an unparser must *reinterpret* when the modified code is unparsed.

For example, the two comments in Figure 3, which the human reader understands as one (following the cue of their vertical alignment) will 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 easily lost.

5.4 The right information

The important goal is to preserve the *right* information, not just all the details. What is right depends on the user and the task. There is some precedent for this in standard compilers. They capture variable names and statement line numbers, not because they are part of the linguistic structure (which in general they are not), but because they are needed for intelligible presentations to humans during debugging.

For the class of tools discussed in this paper, the right information allows an unparser to *reconstruct* a document structure around a somewhat modified piece of code that means the same thing to a human reader as did the original. This implies that more information must be extracted from the original text than has been done in the past.

5.5 Capture abstract documentary structure

Clues to determining the right information are best summarized by the title of Section 4.3: "Relationships matter." The content of comments is important, but only useful when positioned at a particular place in the code at a particular place on the page with a particular arrangement of line breaks around it. If a comment is aligned horizontally with other nearby comments, then that is important too. Here are specific examples of the information that must be captured:

- Where each comment occurred in 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 vertically 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 Figure 12.
- "Extra" syntax, for example redundant parentheses and empty blocks of the sort appearing in Figure 2, especially when associated with comments.

5.6 Extend unparsing rules

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 understanding 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 vertical 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.

5.7 Implementation challenges

Documentary structure, as summarized in Section 4 and captured according to Section 5.5, appears simple and natural, precisely because it is how we, as people, understand documents.

Implementing this strategy, however, presents serious challenges precisely because the compiler-oriented frameworks, in which such tools are built, treat code so differently. For example, the recording of line numbers in compilers is usually implemented by a narrowly defined, special mechanism that lies outside the standard architectural model of compilers; such mechanisms don't often generalize gracefully.

This standard architectural model supports none of the techniques needed to capture documentary structure. For example:

- Comments and white space are typically discarded in the lowest level of a compiler's data flow: between the text stream and the lexical token stream, and 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 automatically generated from a grammar in which they have no meaning;
- There is no place in this simple pipeline model where vertical alignments between adjacent lines can easily be discovered, especially when they involve syntactic constructs; and
- Much of the documentary structure concerns relationships between white space and tokens, but concrete tokens are discarded from syntax trees, leaving no coherent place to record such relationships.

Thus, any tool that successfully preserves documentary structure in the manner described here, will be much more

than a lightly modified compiler plus unparser:

- 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 during the operation of the tools; and
- A more general approach to unparsing must blend unparsing rules of the standard sort with a reconstruction of the original documentary structure.

6. Other directions

Although little has changed in this area for years, 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 none are likely to make the strategy proposed in this paper unnecessary in the near future.

6.1 Literate Programming

The most ambitious and successful attempt to rethink the relationship between documentary structure and source code is based on Knuth's Literate Programming [14]. Knuth also begins with the premise that code is primarily a document for humans, and he takes the radical step of making this aspect paramount. Programmers decomposes code into fragments and embed them in a rich document where prose dominates. Batch tools produce nicely formatted documents (prose with code embedded) for human consumption as well as source code (generally unformatted) for compiler consumption. Knuth and others argue that writing in this style produces better code in the first place [14,18].

Despite 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 isn't 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 available only via batch derivation from a document, so a Literate Program is not easily amenable 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 a language design defect. Source code should 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 [12]. 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." [11] 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.

These suggestions have two drawbacks. First, adoption of new languages, or even commenting conventions, is a rare event. Second, much of the important documentary structure of code, as shown in Sections 2 and 3, is not about syntactic structure at all.

The first widely accepted structural comments appear in the Java programming language [10]. Although enforced only by convention (and, more importantly, by a tool), "JavaDoc" comments are specially tagged and intended to appear only in specific code locations: immediately preceding public class and member declarations. JavaDoc comments are intended to facilitate automatic extraction of interface documentation for hyperlinked publication in HTML [8], but do not apply to the many other uses of documentary structure represented in the conventional formats.

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 claims that syntactic extensions to existing languages can be used, as long as tools hide the complexity from the users [17]. 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 permits greater richness. For example, structure editors such as the Synthesizer Generator represent programs only as annotated syntax trees [20]. Even here, however, comments are seen as little more than anno-

tations on nodes; this has the effect of reducing programmer control over documentary structure without offering anything new in its place.

An even more provocative approach is *hyperprogramming*: representing programs as fully typed persistent language objects that can be manipulated by specialized editors [13]. 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 [6]. Such systems require very different languages and programming infrastructures than are widely available today.

6.4 Make comments unnecessary

A more current trend leads to highly factored object-oriented code, where fine-grained structure, 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."[7]

It is an open question how much documentary structure can be made unnecessary by this approach. It certainly doesn't change the need for intelligent white space layout, nor is it likely to replace the kind of general commentary (explanation, background, and motivation) many readers like to find.

Frequent refactoring is a basic tenet of Extreme Programming [4], and there is a natural interest in helpful tools. These are the kind of language-based transformation tools addressed by this paper, subject to the same requirements. If a programmer refactors with a tool, but also has to investigate and repair comments in every place affected by rippling changes, then the tool will not add value.

7. Conclusions, status, and outlook

This paper proposes that the documentary structure of code (including both comments and white space) is far more important to programmers than one might infer from its treatment by designers of programming languages and language-based tools. Advanced programming tools that perform language-based source code transformations will not be accepted without attention to this issue.

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

Strategies for dealing with documentary structure in language-based transformation tools must ultimately be judged by their success: whether programmers find that the need to repair damage done by tools outweighs their advantages.

Some aspects of the strategy presented here were implemented in 1993 as part of an internal project at Sun Microsystems Laboratories, but in a system that never reached fruition and so could not be evaluated. A new implementation is currently in progress as part of the Jackpot project at Sun Labs, with the expectation that this strategy can be for fully explored.

8. Acknowledgments

Discussions with members of the Jackpot project at Sun Labs, Tom Ball and James Gosling, contributed significantly to this paper, as have many important comments and suggestions from Yuval Peduel. Anonymous reviewers also made helpful comments.

9. Trademarks

References

[1] Paul F. Albrecht, Phillip E. Garrison, Susan L. Graham, Robert H. Hyerle, Patricia Ip and Bernd 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 Dec. 1980, SIGPLAN Notices* **15**,11 (November 1980) 183-193.

[2] Greg J. Badros, "JavaML: A Markup Language for Java Source Code," Ninth International World Wide Web Conference Amsterdam, May 15 - 19, 2000.

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

[4] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley 1999.

[5] Frank W. Calliss, "Problems With Automatic Restructurers," *SIGPLAN Notices* **23**,3 (March 1988) 13-21.

[6] Alan Dearle, Chris Marlin, and Philip Dart, "A Hyperlinked Persistent Software Development Environment," *Proceedings of Hyper-Oz '92: A Workshop on Hypertext Activities in Australia, Adelaide, Australia*, 1992.

[7] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[8] Lisa Friendly, “The Design of Distributed Hyperlinked Programming Documentation,” Sylvain Fraïssé, Franca Garzotto, Tomás Isakowitz, Jocelyne Nanard, Marc Nanard (Eds.), *Hypermedia Design, Proceedings of the International Workshop on Hypermedia Design (IWHD'95) Montpellier, France, 1-2 June 1995*, Springer-Verlag (1996)151-173.

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

[10] James Gosling, William N. Joy, and Guy L. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[11] Peter Grogono, “Comments, Assertions, and Pragmas,” *SIGPLAN Notices* **24**,3 (March 1989) 9-84.

[12] Michael J. Kaelbling, “Programming Languages Should NOT Have Comment Statements,” *SIGPLAN Notices* **23**,10 (October 1988) 59-60.

[13] A.M. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R. Morrison and R.C.H. Connor, “Persistent Program Construction through Browsing and User Gesture with some Typing,” *Proceedings of the 5th International Workshop on Persistent Object Systems (POS5), San Miniato, Italy*, A. Albano, and R. Morrison (eds.), Springer-Verlag, (1992) 86-106.

[14] Donald E. Knuth, “Literate Programming,” *The Computer Journal*, **27**,2 (1984), 97-111.

[15] E. Nygren, M. Lind, M. Johnson, and B. Sandblad, “The Art of the Obvious,” *Human Factors in Computing Systems CHI ‘92 Conference Proceedings, Monterey, CA, USA; 3-7 May 1992*, 235-239.

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

[17] Pierre-N. Robillard, “Automating Comments,” S*IGPLAN Notices* **24**,5 (May 1989) 66-70.

[18] Stephen Shum and Curtis Cook, “Using Literate Programming to Teach Good Programming Practices,” *Proceedings 25th SIGCSE Technical Symposium on Computer Science Education, Phoenix, AZ, February 1994*, 66-70

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

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

[21] Ted Tenny, “Program Readability: Procedures Versus Comments,” *IEEE Transactions on Software Engineering*, **14**, 9, (1988) 1271-1279.

[22] Michael L. Van De Vanter and Marat 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.

[23] Tim A. Wagner, “Modeling User-Provided Whitespace and Comments,” *Practical algorithms for incremental software development environments* Ph.D. Dissertation, Report No. UCB//CSD-97-946 University of California Berkeley, 1997.

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