Stable, Flexible, Peephole Pretty-Printing

Stoney Jackson *

Deptartment of Computer Science and Information Technology
Western New England College
1215 Wilbraham Rd.
Springfield, MA 01119

Premkumar Devanbu, Kwan-Liu Ma

Deptartment of Computer Science
University of California, Davis
One Shields Ave., Davis, CA 95616

Abstract

Programmers working on large software systems are faced with an extremely complex, information-rich environment. To help navigate through this, modern development environments allow flexible, multi-window browsing and exploration of the source code. Our focus in this paper is on *pretty-printing* algorithms that can display source code in useful, appealing ways in a variety of styles. Our algorithm is *flexible, stable, and peephole-efficient.* It is *flexible* in that it is capable of screen-optimized layouts that support source code visualization techniques such as fisheye views. The algorithm is *peephole-efficient*, in that it performs work proportional to the size of the visible window and not the size of the entire file. Finally, the algorithm is *stable*, in that the rendered view is identical to that which would be produced by formatting the entire file. This work has 2 benefits. First, it enables rendering of source codes in multiple fonts and font sizes at interactive speeds. Second, it also allows the use of powerful (but algorithmically more complex) visualization techniques (such as fish-eye views), again, at interactive speeds.

We have built a pretty-printing plug-in for Eclipse that allows the use of sophisticated formatting techniques, including such features as multiple fonts and fish-eye views. Our incremental algorithm enables this plug-in to produce readable layouts (without ugly line-wrapping) within a wide range of window sizes, at interactive speeds.

Key words: pretty printer, software visualiation, development environment, experimental software

1 Introduction

Program maintenance consumes a significant portion of life-cycle costs. The classic Leintz, Swanson, and Tompkins [1] study attributes 50% to 85% of overall lifecycle costs to maintenance. More specifically, source-code reading and comprehension are considered to be major sub-tasks of maintenance activities [2]. Indeed, modern development environments like Eclipse are designed to facilitate navigation and browsing within large software projects. A classic technique to support source-code comprehension is formatting, or *pretty-printing*, whereby visual cues are used in textual rendering to elucidate syntax, task relevance, etc. Indentation and line-spacing are the simplest cues. Modern displays also support color, fonts, underlining, etc.

Pretty-printing is known to promote readability and understanding [3]. Baecker found that formatted code can help programmers answer questions about code up to 25% more accurately [4]. Since programmers spend so much of their time reading and understanding code, improved performance in these tasks can substantially impact software maintenance.

Interactive pretty-printing: Our goal is to promote the use of these techniques in an *interactive* context. Most development time is spent working online within interactive development environments. Thus, pretty-printers that are part of development environments can have substantial impact on the maintenance and development tasks.

Consider a programmer who is browsing through a large software system: she may look at a function, call up a list of all invocations of that function, and then click through that list, successively looking through the different contexts of invocation. At the same time, she may be using a source code visualization that selectively enlarges the displayed text of program fragments data-dependent on the text under her cursor. In this setting, a pretty-printing algorithm can greatly help maintain the readability of code as our programmer browses code and shifts her focus. Most importantly, it must respond quickly, so as not to interfere with our programmer's understanding process.

The Current Approach: To achieve responsiveness, pretty-printers found in most modern development environments are hand-crafted. Their reusability is limited because their implementation is often tangled with that of a particular development environment and that of a particular set of source code visualizations. Also, they can only pretty-print a particular programming lan-

* Corresponding author.
Email addresses: `hjackson@wnec.edu` (Stoney Jackson), `devanbu@cs.ucdavis.edu` (Premkumar Devanbu), `ma@cs.ucdavis.edu` (Kwan-Liu Ma).

guage; that is, they are *language dependent*. Lacking reusability, much effort must be expended to build pretty-printers for different languages, development environments, and source code visualizations, incurring all the development costs for each variation. *We propose a different algorithmic approach, peephole pretty-printing, which can enable the use of language*-independent *techniques in an interactive context.*

The literature contains many examples of language-independent pretty-printers [5–13], and software frameworks that ease the development of pretty-printers for different languages [5,12,14–16]. However, the pretty-printers they produce are either not responsive enough for the environments which we have described above, are not flexible enough for use with different source code visualizations, or do not make efficient use of display space.

Our Approach: In this paper, we introduce a novel pretty-printing algorithm that makes good use of display space, *and* is flexible, language-independent, and responsive. While the performance of existing pretty-printers is a function of a file's length, ours is a function of the relatively small window size to which we format files. As a result, our algorithm can pretty-print a typical window of code in 139ms, whereas a representative of existing pretty-printers takes 1700ms: which is an order of magnitude gain in responsiveness. Our work, then, enables the incorporation of sophisticated pretty-printing techniques in a wide variety of interactive development environments. In fact, we have incorporated our approach into the Eclipse IDE [1] . This work can be downloaded from `http://peephole.cs.ucdavis.edu/`. While the tool is still a prototype and is evolving, the reader can experience the fast peephole rendering of the visible portion of the window; certainly we invite feedback.

The outline of the paper is as follows. We begin in Section 2 with some technical background on pretty-printers. In Section 3, we present our peephole algorithm. In Section 4, we present performance data that supports our claim of peephole-efficiency, and some samples of rendered text. In Section 5 we present comparisons to more closely related work.

2 Background

Pretty-printers began as hand crafted tools, designed to format code for a particular programming language in a specific style. While some evidence exists in favor of certain broad rules, such as using indentation to show the logical, nested structure of code [17], in general, style is often a matter of programmer preference [18]. This led to the construction of pretty-printers that allow

[1] Eclipse is available at `http://www.eclipse.org`.

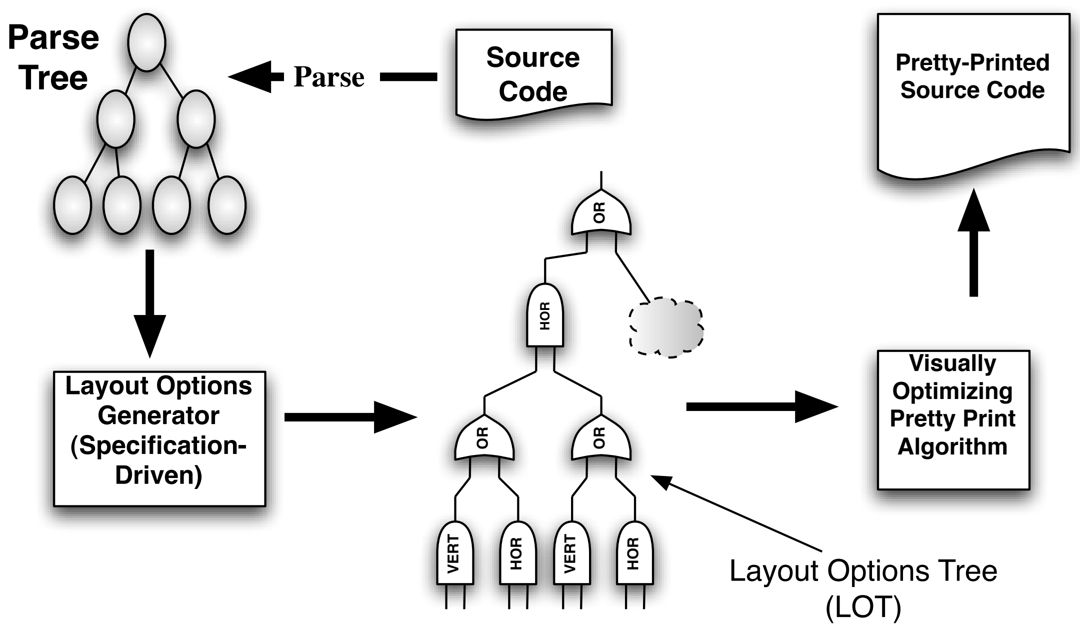

Parse Tree
Parse
Source Code
Pretty-Printed Source Code
OR
HOR
Layout Options Generator (Specification-Driven)
OR
OR
Visually Optimizing Pretty Print Algorithm
VERT
HOR
VERT
HOR
Layout Options Tree (LOT)

Fig. 1. Space-optimizing pretty printers work by first parsing source code into an abstract syntax tree (AST). Then a layout options generator (LOG) generates a layout options tree (LOT), based on a formatting specification. Then a visually optimizing pretty-printer produces the final pretty-printed text for a given display width.

programmers to customize the format produced by a pretty-printer to suit their own preferences [4,19,20]. However, pretty-printers were still programming language-specific, and had to be hand-written for each language and dialect.

Oppen [21] was among the earliest to invent a language independent pretty-printer that is sensitive to the width of the output medium. Following Oppen's work, several language-independent pretty-printers are described in [22,5–13]. Each varies slightly in expressivity and complexity; however, their basic design is the same. Figure 1 illustrates the operation of a typical modern pretty-printer. Most are space-optimizing: they try to make best use of the available vertical and horizontal space, trying to avoid both blank areas, and ad-hoc line-wrapping. A programmer specifies how to pretty-print a program in a formal specification language. The specification describes the different options for printing each node type of an abstract syntax tree (AST): *e.g.,* different ways of indenting and spacing an `if ...then ...else` statement, depending on available width. A layout options generator (LOG), driven by this specification, produces a layout options tree (LOT). This tree encodes the different options for laying out the program text. Given a LOT, and a specific screen width, the pretty-print algorithm selects an optimal layout for that width from all the options implicit in the LOT. Because LOTs encode all possible layouts for some code, they can be quite large (see 5 for a more detailed discussion about LOT sizes). For this reason, LOGs usually produce portions of LOTs lazily, i.e., as the pretty-printer needs them.

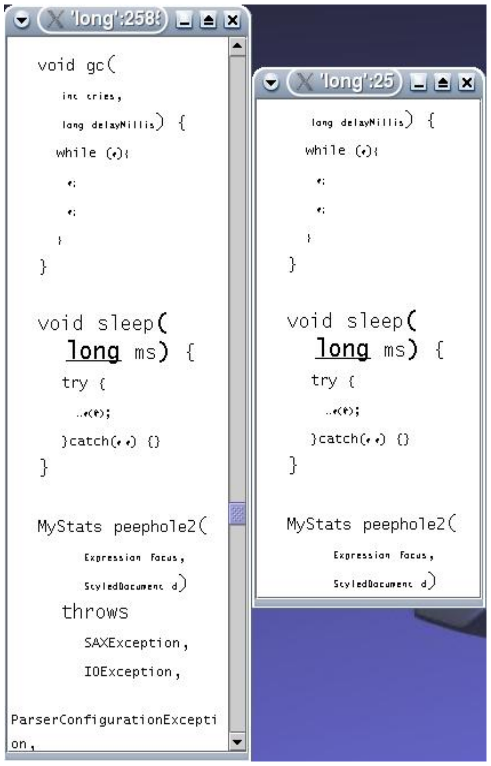
There are many software frameworks for building pretty-printers [5,12,14–16]. At their core is some variation of Oppen's pretty-printer technique. These frameworks ease the construction of language specific pretty-printers by allowing format descriptions for a target language to be specified in a common, high-level, formatting language. Specifications written in this format language drive the LOG, and include all the rules for proper syntax-driven nesting and indentation; so the LOT will include the proper indentation context for every possible layout that it encodes. Programmers can customize the pretty-print style by modifying layout specifications. These language-independent space-optimizing pretty-printers, however, have an undesirable property: to print a particular line in a file that may be of interest, they have to compute the layout of *all the preceding lines* of text. This property reduces responsiveness in interactive settings such as IDEs, where well-laid out program text can improve reading efficiency and productivity.

This paper describes how we augmented a LOT structure and pretty-print algorithm framework to achieve the responsiveness necessary for their use in interactive development environments. Thus, from these systems we would gain language independence, customizability and reusability, while our techniques adds the desired responsiveness and additional flexibility.

We achieve responsiveness through *peephole-efficiency.* A *peephole-efficient* algorithm only does work proportional to the amount of text that is actually visible to the user, *i.e.*, the useful text-display region of the window (see Figure 2). Peephole-efficiency is a practical way of achieving responsiveness since the number of lines in the average source file is often more than what the average window can display. Consider the Java code base for the libraries in Java SDK 1.4.2. It contains 4,126 Java files. The average file is 650 lines of code formatted to an 80-character width. If we were to view such files in a window that can display 100 lines of 80-column-width code, [2] a peephole-efficient pretty-printer would only need to format less than 16% of the file on average. In practice, we expect this average to be significantly smaller. For one, we expect programmers to spend more time understanding larger, more complex files than smaller, less complex ones. Thus, our pretty-printer will more often be used on larger files. Also, we expect most windows to be less than 80 characters wide and 100 lines long, further reducing the work performed by a peephole-efficient pretty-printer.

Perhaps the largest obstacle to achieving peephole-efficiency is this: the proper layout of a source code fragment is *not a local property.* Given a particular fragment of code, its actual positioning on a display medium includes 1) the

[2] A 1200 pixel high window can display about 100 lines of code in a 10 point font (1 point $\approx$ 1 pixel), given a standard 2 point space between lines (the leading).



```
void gc(
    int tries,
    long delayMillis) {
   while (•){
     •;
     •;
   }
}

void sleep(
    long ms) {
   try {
     ..•(•);
   }catch(• •) {}
}

MyStats peephole2(
       Expression focus,
       StyledDocument d)
   throws
       SAXException,
       IOException,
ParserConfigurationExcepti
on,
```

```
    long delayMillis) {
   while (•){
     •;
     •;
   }
}

void sleep(
    long ms) {
   try {
     ..•(•);
   }catch(• •) {}
}

MyStats peephole2(
       Expression focus,
       StyledDocument d)
```

Fig. 2. The peephole layout, on the right is *stable* since its layout is identical to that of its counterpart in the global layout, on the left.

indentation of the fragment and 2) whether there is a line-break in the middle, beginning or end of the fragment, etc. These are typically dependent upon 1) the source fragment's syntactic role in the overall program, 2) the window width, and 3) the manner in which the program surrounding it is displayed. In general, the position of a particular token may depend on the position of all other tokens in the document. We will restrict ourselves to algorithms that have a finite lookahead (e.g., never looking ahead more than x characters). Even these algorithms, the position of the n^{th} token depends on the position and size of the $(n-1)^{th}$, and thus inductively upon the previous set of $n-1$ tokens. A peephole-efficient algorithm should respect these non-local dependencies.

A *stable* peephole pretty-print algorithm produces the same view in a window as would be produced by an algorithm that renders the entire file. Figure 2 illustrates this idea. The two windows contain code from the same source file. The left window was formatted by a non-peephole pretty-printer that formats the entire file. The right contains code formatted by a peephole pretty-printer. The left window has been scrolled to show the corresponding code fragment that was formatted in the right. Notice that their format is identical. In this case, we say that the peephole's layout is stable. If a peephole pretty-printer always produces stable layouts, we say it is stable as well. If a pretty-printer is not stable, then the formatting of the rendered source-code might abruptly change as a programmer scrolls or shifts her focus in a peephole view. This would be a distraction and a hindrance to program comprehension. The core intuition of our algorithm is the identification of an *anchor* which allows us to partition the layout of code into non-interfering segments, thus achieving *stability*. The challenge, then, is to exploit the invariants in the LOT to find an anchor as quickly as possible, so that layout can be performed at interactive speeds.

We add additional *flexibility* to existing frameworks by supporting dynamic elision techniques, such as Fisheye views [23], and source code visualizations that alter fonts and font sizes, such as Seesoft views [24]. These techniques help programmers cope with the contextual complexities and large scale of modern software systems.

The combination of peephole-efficiency, stability, and flexibility enables the incorporation of a practical, customizable, useful pretty-printing tool into Eclipse.

3 Designing the Algorithm

Pretty-printers can be viewed as instances of the more general class of document-formatting tools, such as LaTeX. Certainly, the issue of "peephole" viewing also arises, in the more general context. For example, Chen *et al* [25,26] discuss the problems that arise in building an efficient, incremental processor specifically for TeX. Many of these problems are also relevant to peephole pretty-printing. Chen [25] advocates an "augmentation approach" to building peephole formatters: begin with an existing (non-peephole) formatting algorithm, identify ways in which its work can be partitioned; then augment it with an incremental work-load manager that invokes the existing algorithm selectively to layout just the portion of text that is needed. We follow this approach: we began with Wadler's pretty-printing algorithm [13], and carefully analyzed his algorithm for "articulation points" where we could partition the work it does. Thus, we begin with a brief description of the data structure and algorithm originally presented in [13].

3.1 The Algorithm Without Peephole

The LOT As shown in Figure 1, the layout options tree (LOT) is generated from an abstract syntax tree (AST) by a layout options generator (LOG). Recall that a LOT encodes all the possible layouts for some source code, and that a pretty-printer algorithm will select a layout from a LOT that is optimal for a given width. The LOT structure we will use has 5 different types of nodes:

`Text(`*string*`)`: Represents a sequence of characters that contains no new-lines.

`Break`: Represents a new-line followed by some indentation, where the indentation amount is the sum of the indentation amounts in the `Break`'s ancestral `Nest`s.

`Nest(`i`,`x`)`: Adds an additional amount of indentation i after every `Break` in x. The effects of `Nest`s are cumulative.

`Cat(`x`,`y`)`: Concatenates documents x and y.

`Choice(`x`,`y`)`: Represents a choice between two alternative layouts of the same document.

A LOT is a tree composed of these node types. For example:

```
Choice(Text("Hello, Ma"),
       Cat(Text("Hello,"),
           Nest(4, Cat(Break, Text("Ma")))))
```

encodes two possible layouts the document "Hello, Ma": namely,

```
Hello, Ma       and     Hello,
                            Ma
```

Excluding `Choice`, the other four constructors represent compulsory layouts, *i.e.*, the pretty-printing algorithm has no choice in how it produces the output. There are two invariants that govern `Choice`s that are central to the design of our peephole optimization. These invariants are enforced by the layout options generator (LOG) (see figure 1).

Invariant 1: *The left and right sides of a* `Choice` *represent the same fragment of code.*

Invariant 2: *The first line of the left layout is at least as wide as the first line of the right layout.*

As a result of the first `Choice` invariant, LOTs can become quite large. Each child of a `Choice` represents an alternate layout of the same code. More, `Choice`s may be nested. In the worst case, the size of a LOT is approximately $O(2^n)$, where n is the size of the original source code file. In [13], Wadler avoids this space blowup by using constructing portions of LOTs lazily. Generally, the left side of a `Choice` is a function of the right. That is, the left side can be described in terms of the right. Thus, we only need to store the right sides of `Choice`s, and generate the left sides lazily as needed. Wadler implemented the LOT data structure and pretty-print algorithm in the lazy, functional language of Haskell [13]. Our implementation is in the eager, object-oriented language of Java. Since we need to construct LOTs lazily to avoid exponential space consumption, we employed several known lazy object-oriented patterns [27].

This paper assumes that the left side of a `Choice` is always a function of the right. We also assume that this function can be evaluated lazily, and that given a node on the left side, its counterpart in the right is known. It is also important to the performance of our algorithm that the functions be near linear in time with respect to the size of the right hand side. All layout alternatives described in the literature can be defined as a function with these properties.

The five LOT constructors presented above are quite expressive, and can capture many different layouts. A well-written LOG specification will produce a good set of different layout options in the LOT, which will then allow the pretty-printer to generate attractive layouts for a wide range of display widths.

The Pretty-Print Algorithm The pretty-print algorithm selects from a LOT the "best" layout for a given width. "Best" is defined by Wadler [13] as the layout that produces the fewest line breaks and still fits within the given width. Unfortunately, searching the entire LOT, which has $O(2^n)$ nodes, is intractable. Wadler [13] uses a greedy algorithm that produces nice, reasonably space optimized layouts in $O(n^2)$ worst case time.

The algorithm performs a depth-first traversal of a LOT, starting at the first token in a file (since it's not peephole). It outputs `Text`s and `Break`s as they are encountered. When a `Choice` is encountered, the algorithm chooses between the `Choice`'s left and right layouts. It tries to select the layout with the fewest `Break`s that still fits in the given width without examining too much of the LOT, *i.e.*, without looking too far ahead. By the second invariant, the algorithm knows that the first line of the left layout is at least as wide as the right. That is, by taking the left layout, the algorithm may be able to use more of the width, and delay the output of a new-line. So, if the first line of the left layout fits in the remaining width of the current line, the algorithm selects the left layout and discards the right. Alternatively, if the first line of the left does not fit, it takes the right and discards the left.

To make this decision, the algorithm must track the remaining width for the current line. A `Break` starts a fresh, indented line. So the remaining width is set to the full width minus the current indentation amount whenever a `Break` is encountered. The current indentation amount is updated whenever the algorithm enters or leaves a `Nest`. `Text`s subtract their width in the current font from the remaining width.

3.2 Supporting Holophrasting and Distortions

While pretty-printing certainly improves readability, large and complex programs present another challenge for programmers, specifically, *scale* and *context*. *Holophrasting* is the collective name of a set of techniques, including Furnas's "fisheye" views [23], that uses various types of "interest metrics" to select a subset of source code fragments that are relevant to a given focus statement selected by a programmer. Interest metrics can be used to either simply elide text, or to visually condition it in a more finely graduated manner, *i.e.,* using larger or more emphatic fonts for more relevant code.

To support holophrasting, we extend the above algorithm and data structure as follows. LOT nodes may be marked "elided" at runtime. Before processing a node, the pretty-print algorithm first checks if the node is to be elided. If the node is marked elided, then the node and its subtree are replaced with a predefined placeholder, such as ellipses.

The check may be a constant lookup in a predefined table, or it may be a more complex computation performed at runtime. We abstract this implementation detail from our algorithm by presenting the algorithm with an interface for checking the elision property of a node. Through this interface, the pretty-print algorithm communicates with a holophrasting engine that implements the holophrasting technique currently being used. This separation also makes it possible to reuse our algorithm with different holophrasting techniques and implementations.

3.3 Achieving Peephole-Efficiency

Our goal in this research is to enable the use of both pretty-printing (with multiple fonts, etc.) and holophrasting *at interactive speeds*. Our approach is peephole printing, which renders only as much text as is neeeded to fill a window (the peephole) around the current focus of the user. However, we want the algorithm to produce the same rendering that would be produced by a pretty printer that (more slowly) rendered the entire file.

The first time a peephole is requested for a file, the file must be read into memory, parsed into an AST, and a LOT for the AST generated. Once this has been done for a file, subsequent peephole requests do not require this overhead. In exploratory environments with dynamic visualizations, we expect the latter to be the common case. Therefore, for the remainder of our discussion, we assume that a file's AST and LOT have already been built.

Our approach is based on the intuition of an *anchor*. An *anchor* is a line of text in the final output which is guaranteed to occur on a new line, with a known indentation for a given width. In our LOTs, an *anchor* is a `Break` that is guaranteed to be outputted by our pretty-print algorithm for a given width.

The layout of the text that comes after an anchor is independent for what comes before for a given width. This is because an anchor is known to be present in the output for that width, and it starts a new line at a known indentation. Therefore, once we have identified an anchor, we know where to position subsequent text. It follows that, for a given width, the layout that is produced by starting at the anchor is identical to that produced by starting the pretty-printer at the root of the LOT. This is how our algorithm achieves *stability*.

Assuming we know how to identify anchors, we can pretty-print a peephole about a focus for a given width as follows. Starting at the focus in the LOT, we search for anchors working backwards through the LOT. Once we have found an anchor, we simply pretty-print forward through the focus until we fill the peephole. If no anchor is found, we pretty-print from the root of the LOT through the focus until we fill the peephole.

So the question becomes, how do we find anchors quickly?

Finding an Anchor To identify an anchor in our LOT, we must find a break that, even if it is under a choice node, given a width, will definitely result in a new line in the output. There are two reasons a `Break` may be discarded: 1) the `Break` or one of its ancestors is elided, or 2) for some ancestral `Choice`, the pretty-printer selected the child that does not contain the `Break`. We must ensure neither of these are true to prove that a `Break` is an anchor.

For elision, the test is simple. The holophrasting engine is consulted to see if the `Break` or any of its ancestors are to be elided. If any are to be elided, then the `Break` is not an anchor.

Next, we must ensure that all the `Choice` nodes above this node did not choose the alternative to the `Break` under consideration. For each `Choice`, there are two possible outcomes: the right is taken, or the left is taken.

For the left to be taken, the first line of the left layout fits in the *remaining width* of the current line. The remaining width can only be determined by starting the pretty-printer at some earlier position, which is the problem we are trying to solve in the first place. Because we can never prove that a left side is taken, we never try to identify anchors on the left side of any `Choice`.

For the right to be taken, the first line of the left layout must exceed the remaining width. Again, we do not know the remaining width. However, we do know the *full width*; and, if the first line of the left exceeds the *full width*, then it will never fit in any remaining width. In this case, we know that the left side will *never* be taken. So, if a `Break` is on the right side of all ancestral `Choice`s, and all of these `Choice`s will take their right side, then the `Break` may be an anchor.

As an optimization, during the construction of the LOT, we store with each `Choice` a reference to the immediately preceding anchor candidate. This allows us to quickly step to the previous anchor candidate and skip over other `Break`s that are right descendents of the failed `Choice`. If none of the candidates qualify, then we must start formatting from the beginning of the file.

This method of selecting anchors seems highly conservative, *i.e.,* too pessimistic or picky in selecting a possible anchor. This might suggest that we would reject many perfectly good anchors and pick one that is far away from the focal point, thus doing a lot of needless work. In practice, however, we find very good anchors quickly (see Section 4). The reason is simple: most reasonable pretty-printing specifications will "strongly encourage" line breaks at certain frequently-occurring locations, e.g., the top level statements in a method body. Our heuristic for finding anchors, conservative as it is, quickly zeroes in on one of these recurring line breaks. However, it is important to note that *our algorithm is independent of both the programming language and the layout style embodied in the pretty-printing specifications; no reprogramming of the peephole pretty-printing algorithm itself is required for a different language, or for a different pretty-printing style specification.*

Printing from an Anchor Once an anchor is found, we can print forward from there until the focal point is reached, and then proceed further till the entire visible region is filled. This is done using the standard, non-peephole pretty-printer. Essentially, we need to "position" the pretty-printer at the anchor, and set its state so that it believes that it has arrived at that point in the usual way: and then just start it up. Recall that the algorithm essentially performs a depth-first traversal of the LOT. To restart the algorithm at an arbitrary position in the LOT, its stack must be reconstructed as if it arrived at the node during a normal traversal. This can be done by a single walk from the root of the LOT to the node from which to restart. This process is $O(h)$, where h is the height of the LOT. For reasonable layouts, this height can be expected to grow roughly linearly as the depth of the AST. Thus, if the program is well-structured, the LOT depth should grow slowly as compared to the program size.

As described, the peephole algorithm may place the focal point anywhere within the peephole. With slight alterations to the algorithm, the focus can be maintained at a desired location on the screen. For brevity, these alterations are not described here.

It is also possible for the algorithm to print more than the window size. We measured the number of extra lines the algorithm formatted across 45 sample runs with varying window sizes and various fisheye degrees, and found that typically no more than 2 extra lines were formatted.

In rare cases, the algorithm formatted 7 extra lines. This excess work is negligible in the context of programs that are significantly larger than the size of the peephole.

3.4 Implementation

We have implemented two different pretty-printing systems, both using our peephole algorithm. The first implementation was a prototype used to empirically evaluate the performance of our algorithm (the results of which can be found in Section 4). The second was a plug-in for the Eclipse development environment. This section briefly describes their implementations and the challenges we encountered along the way.

The Peephole Pretty-Printing Algorithm The implementation of our pretty-print algorithm, which is used in both the prototype and our plug-in for Eclipse, is a Java implementation of Wadler's pretty-print algorithm [13] augmented to support elision techniques and (re)starting from an arbitrary node in a LOT. The structures and strategies used to implement the lazy behavior that are essential for an efficient implementation of Wadler's algorithm are based on those described in [27].

The Prototype Our initial prototype uses JavaML [28] to parse (offline) Java code into XML abstract syntax trees. It uses Xerces2 Java Parser [3] to parse the XML form of the AST into a DOM tree. We custom-built a layout options generator (LOG) that traverses JavaML DOMs and produces LOTs. For reasons discussed below, comments are not represented in JavaML ASTs; as a result, the views produced by this prototype lack comments. Comments are addressed in our plug-in implementation for Eclipse.

The Eclipse Plug-In Our implementation for Eclipse is similar to the prototype, but uses Eclipse's Java Development Toolkit to parse code (online) into ASTs. Again, we hand crafted a LOG to walk ASTs and produce LOTs.

[3] `http://xml.apache.org/xerces2-j/index.html`

The LOG is used whenever a file is opened for the first time or when a file is changed. Otherwise, once a LOT has been built, it is reused to produce views of source formatted to the current window's width.

As discussed in Section 2, the LOG controls the style of code through its construction of LOTs. In a more general framework, LOGs would be generated from high level specifications that describe desired stylings. Such a framework allows users to easily customize formats to reflect their personal coding style. At the time of writing, styling information is hard coded in a hand-crafted LOG. In the future, we plan to adapt an existing LOG generator as a front end to our existing tool.

Comments Comments are a challenge to any system that reproduces code from an AST, because comments are not typically represented in ASTs. The trouble is that most programming languages allow a programmer to insert comments almost anywhere without indicating the artifact she is commenting. So, parsers cannot easily determine a programmer's intent, and therefore cannot determine the AST node to which to attach a comment.

To handle comments, our plug-in's LOG uses an approach similar to that first described in van den Brand and Visser's work [12]. JDT produces AST nodes that identify the region of source code from which they were built. Using this information, our LOG extracts comments by scanning the gaps between AST node regions. Our LOG does not try to associate comments with AST nodes, but instead determines the spacing between a comment and a token based on the original spacing and the comment's type using three heuristic rules: First, if a comment and an adjacent token are on the same line, or are separated by exactly one new line character, then an optional line is inserted. Thus, when a window is sufficiently wide, the two will appear on the same line. Otherwise, they are placed on separate lines. Second, If a comment and an adjacent token are separated by more than one line, those lines are not considered optional and are maintained by the LOG. Finally, a line comment is always followed by at least one line.

The implementations described above along with examples are available on our website (c.f. section 6).

4 Results

We now present some performance data comparing our peephole technique with a global display approach, which lends support to our claim of peephole-efficiency and flexibility. We also present some sample outputs that support our claim of stability.

The data reported is on an earlier, stand-alone implementation of our peephole algorithm. This implementation allowed us to isolate and measure differentially the performance of our pretty-printing algorithm in various settings of interest, without interference from a complex embedding platform such as Eclipse. We believe these measurements reveal precise characterstics of our algorithm, and its performance under different conditions.

We consider 3 different independent variables: the width of the display window, the height of the display window, and the degree of selectivity imposed. Our dependent variable is the pretty-printer execution time.

The times presented here measure the time for the algorithms to fill a peephole about a focus. They do not include the time to parse code into an AST. Nor do they include the time to construct LOTs from ASTs. However, they do include the time required to lazily construct portions of the LOTs. Likewise, they include the time necessary to calculate fisheye value when fisheye is used. This scenario is representative of that of a user browsing code where ASTs, and therefore LOTs, rarely change.

To reduce errors in our measurements, we ran each combination of independent variables 285 times, changing the location of the peephole (and the focus for the fisheye) each time, and averaged the dependent variable. This gives us a 90% confidence that our measured average value is within 5 milliseconds of the actual average. Although the data shown in this section is for one 2000 line Java file, we also ran our algorithm on several different files of different sizes as a sanity-check to confirm that the data was in the same range, and showed the same trends.

The global display approach lays out the entire document up to end of the visible region. In general, it would not need to layout the entire file. So for the purposes of comparison, we take 50% of the actual time needed to layout the entire file.

During the timing measurements, the algorithms write their output to a Java Swing `StyledDocument` object, which is an off-screen buffer that can be posted to an interactive Swing window. Layout stability was manually checked in a separate run: random samples of the `StyledDocument` objects were posted to Swing windows for visual inspection.

All experiments were performed using Sun's Java Runtime Environment, version 1.4 on a dual Intel Xeon, 2GHz processor machine with 1GB of memory.

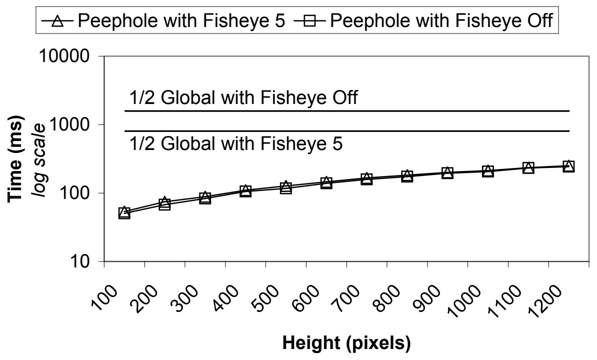


Fig. 3. Variation of performance with height of display window, in pixels. Roughly 15 pixels is a line, and 1200 pixels is 80 lines.

Varying Window Height Figure 3 shows how the global and peephole algorithms vary as the height of the window changes. The window width was fixed at 480 pixels, about 70 characters. Roughly 15 pixels make a line; 1200 pixels is the height of most displays. At a sample height of about 40 lines (600 pixels), with fisheye turned off, the peephole algorithm finishes its work in 139 milliseconds. In this case, the peephole version shows a significant advantage (139ms vs. 1700 ms), and is arguably within the acceptable limits of interactive speeds for browsing behavior. With the fisheye, this performance worsens slightly to 145 ms.

Varying Window Width Figure 4 shows how the performance of the algorithms varies as a function of width. The height was fixed at 300 pixels, which is approximately 19 lines. Width is increased from 100 pixels to 1600 pixels (one character is about 7 pixels wide). As the width increases, the layout time increases slightly for both algorithms. The pretty-printing algorithm tries to fit as much text as possible on each line, subject to line-breaking requirements given in the LOT. A typical pretty-printer specification (ours does) might say "put all the lines at the top level of a function body in one line, or put them all on separate lines". Since all of a function's code usually will not fit into one line, the latter choice is generally inevitable. However, the pretty-printer does have to consider the first possibility, expending effort proportional to the width of the line.

Varying Fisheye Degree The fisheye degree controls the amount of code that is visible. As the fisheye degree is increased, more of the file becomes visible. Figure 5 shows how the layout time varies for the global and peephole techniques. The width and height of the peephole window was fixed at 480 and 300 pixels respectively. For small fisheye values, the times taken by the two algorithms are nearly the same. This is because most of the source file is being elided, leaving behind a small enough remnant that nearly fits in the

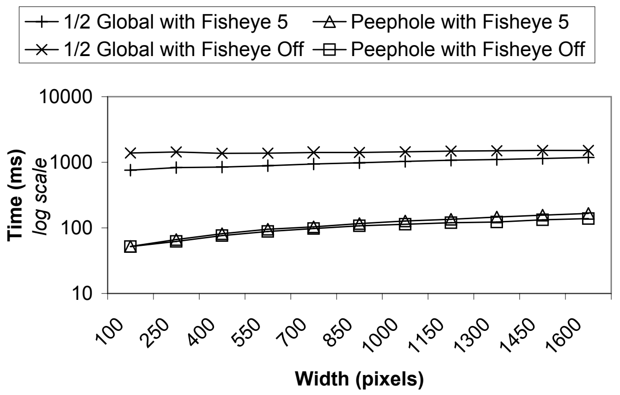


Fig. 4. Variation of performance with width of display window, in pixels.

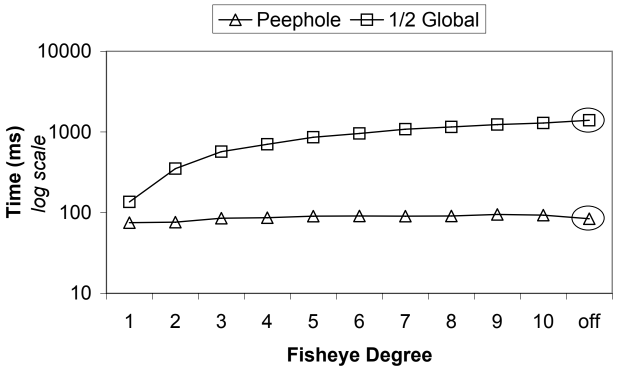
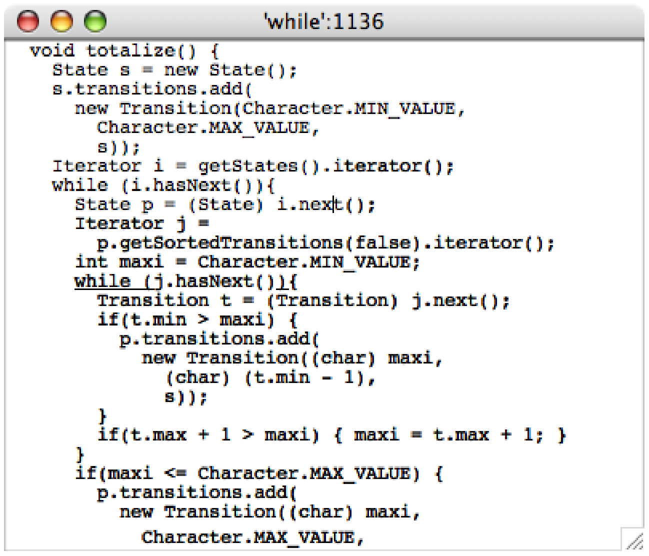


Fig. 5. Variation of performance with degree of selectivity of fisheye. Higher numbers are less selective. Global is entire file; peephole is a 480 × 300 pixel window.

given window size. So the work performed by global pretty-print is not much more than that performed by peephole. As the degree is increased, the unelided remnant grows, and thus global takes more time lay it all out. However, the peephole is rendering a more or less constant amount of text, and remains relatively constant. This supports the claim that, even when augmented with a fisheye feature, our peephole pretty-printer only does work roughly proportional to the size of the window.

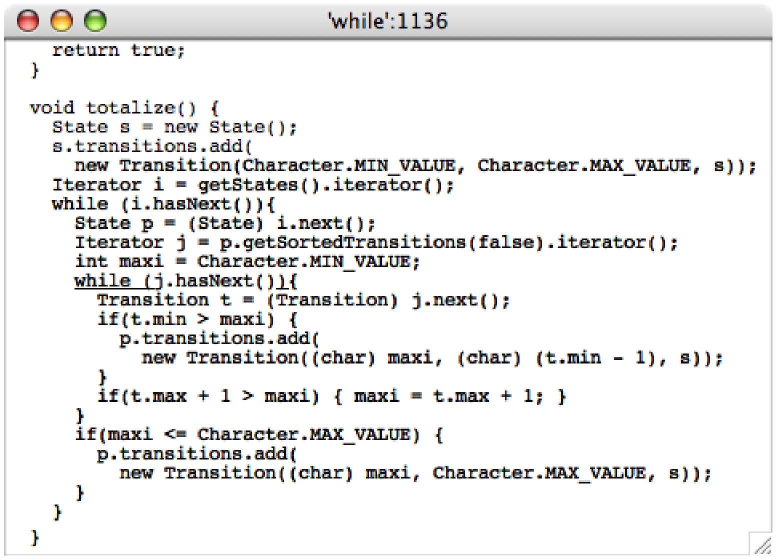
Examples We now present some examples of the peephole pretty-printer in action at various widths and fisheye degrees. The goal of these examples (presented with execution times) is to demonstrate the usefulness of combining space-optimizing layouts with fisheyes. It can also be seen, thanks to the peephole optimization, that the execution times are arguably within the acceptable range for refreshing windows.

In our first example, shown in Figure 6, code has been formatted to a 400 × 300 pixel peephole with a focus on the keyword `while`. Execution time for displays of this size averages 80 ms. The window shown in figure 6 has



```
void totalize() {
  State s = new State();
  s.transitions.add(
    new Transition(Character.MIN_VALUE,
      Character.MAX_VALUE,
      s));
  Iterator i = getStates().iterator();
  while (i.hasNext()){
    State p = (State) i.next();
    Iterator j =
      p.getSortedTransitions(false).iterator();
    int maxi = Character.MIN_VALUE;
    while (j.hasNext()){
      Transition t = (Transition) j.next();
      if(t.min > maxi) {
        p.transitions.add(
          new Transition((char) maxi,
            (char) (t.min - 1),
            s));
      }
      if(t.max + 1 > maxi) { maxi = t.max + 1; }
    }
    if(maxi <= Character.MAX_VALUE) {
      p.transitions.add(
        new Transition((char) maxi,
          Character.MAX_VALUE,
```

Fig. 6. Code formatted to a 400 × 300 pixel window with a focus on the keyword `while` and fisheye off.



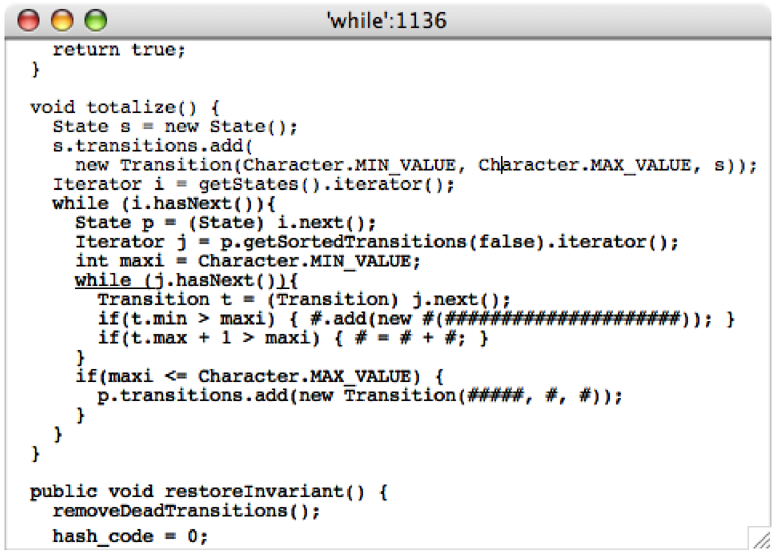
```
    return true;
  }

  void totalize() {
    State s = new State();
    s.transitions.add(
      new Transition(Character.MIN_VALUE, Character.MAX_VALUE, s));
    Iterator i = getStates().iterator();
    while (i.hasNext()){
      State p = (State) i.next();
      Iterator j = p.getSortedTransitions(false).iterator();
      int maxi = Character.MIN_VALUE;
      while (j.hasNext()){
        Transition t = (Transition) j.next();
        if(t.min > maxi) {
          p.transitions.add(
            new Transition((char) maxi, (char) (t.min - 1), s));
        }
        if(t.max + 1 > maxi) { maxi = t.max + 1; }
      }
      if(maxi <= Character.MAX_VALUE) {
        p.transitions.add(
          new Transition((char) maxi, Character.MAX_VALUE, s));
      }
    }
  }
```

Fig. 7. Code formatted to a 480 × 300 pixel window with a focus on the keyword `while` and fisheye off.

been stretched in height to show all the work performed by the pretty printer, including the extra lines that are sometimes formatted, as described in section 3.3. This is true for the other figures in this section.

In the second example, shown in Figure 7, we have widened the peephole to 480 × 300 pixels maintaining the same focus. Execution time for formatting windows of this size averages 85 ms. Notice that with the wider peephole some of the code has been folded into one line. This packs in more text, revealing more of the surrounding code.



```
  return true;
}

void totalize() {
  State s = new State();
  s.transitions.add(
    new Transition(Character.MIN_VALUE, Character.MAX_VALUE, s));
  Iterator i = getStates().iterator();
  while (i.hasNext()){
    State p = (State) i.next();
    Iterator j = p.getSortedTransitions(false).iterator();
    int maxi = Character.MIN_VALUE;
    while (j.hasNext()){
      Transition t = (Transition) j.next();
      if(t.min > maxi) { #.add(new #(######################)); }
      if(t.max + 1 > maxi) { # = # + #; }
    }
    if(maxi <= Character.MAX_VALUE) {
      p.transitions.add(new Transition(#####, #, #));
    }
  }
}

public void restoreInvariant() {
  removeDeadTransitions();
  hash_code = 0;
```

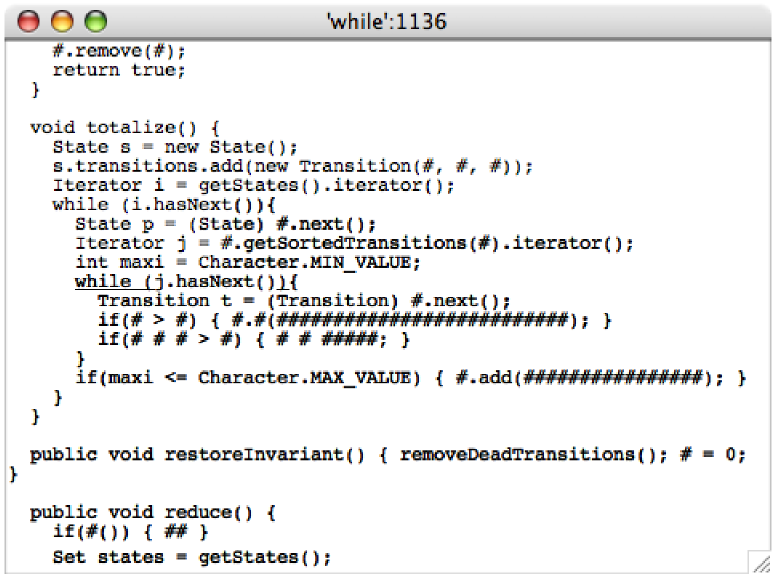
Fig. 8. Code formatted to a 480 × 300 pixel window with a focus on the keyword `while` and a fisheye degree of 7.

In a third example, shown in Figure 8, the same code has been formatted in the same size window as our previous example and with the same focus, but now fisheye has been turned on at a degree of 7. Execution time for similar windows averages around 90 ms. Notice that several long parameter names have been elided in the call to `transitions.add`; this reveals some of the next method `restoreInvariant`.

In our last example, shown in Figure 9, the same code has been formatted to the same size window as our previous two examples and again uses the same focus, but now the fisheye degree has been narrowed to 5. Such displays are formatted in 90 ms on average. Narrowing the fisheye degree has caused more elision and more folding of lines, making all of `restoreInvariant` visible and more surrounding code visible in the peephole.

5 Related Work

There is much existing work on pretty-printers [4,6–10,12,13,15,16,21,29,30]. They do not address pretty-printing for width sensitive, interactive programming environments as those we have described—their concern is generally rendering entire files in visually static environments. Conservative pretty-printing [31] is concerned with pretty-printing files while retaining as much as possible of the user's original textual layout. Our pretty-print algorithm is based on Wadler's work [13] because of the elegance and expressivity of its format specification language. By demonstrating our algorithm on his library, we expect our algorithm can be adapted to other LOT structures as



```
    #.remove(#);
    return true;
  }

  void totalize() {
    State s = new State();
    s.transitions.add(new Transition(#, #, #));
    Iterator i = getStates().iterator();
    while (i.hasNext()){
      State p = (State) #.next();
      Iterator j = #.getSortedTransitions(#).iterator();
      int maxi = Character.MIN_VALUE;
      while (j.hasNext()){
        Transition t = (Transition) #.next();
        if(# > #) { #.#(###########################); }
        if(# # # > #) { # # #####; }
      }
      if(maxi <= Character.MAX_VALUE) { #.add(################); }
    }
  }

  public void restoreInvariant() { removeDeadTransitions(); # = 0;
}

  public void reduce() {
    if(#()) { ## }
    Set states = getStates();
```

Fig. 9. Code formatted to a 480 × 300 pixel window with a focus on the keyword `while` and a fisheye degree of 5.

well. Similarly, since the pretty-printer frameworks described above use a format language at their core, we expect our work to allow them to generate peephole-efficient pretty-printers for interactive environments.

As mentioned in section 2, existing pretty-printers vary in the expressivity of their format language. A pretty-printer must balance expressiveness with performance. For example, although fairly expressive, the BOX format language [22] one cannot correctly express a format that properly wraps long line comments for a C-like language, which require a line comment token (//) to be inserted after each new line. However, the LOTs produced from BOX specifications are linear in size with with respect to the size of the original document. Alternatively, it is possible to correctly specify line comment wrapping in Wadler's format language [13]. The sacrifice, of course, is that LOTs may become superlinear in size with respect to the size of the original document. To compensate, systems such as Wadler's use lazy evaluation to tame the size blow-up of LOTs.

Some early work [32,33] describes pretty-printers that, given a focal position in some source code and the dimensions of a window, uses formatting and elision to minimize the unused window space, and maximize the amount of "useful" code displayed. The layout produced is locally optimal, but is not stable. This leads to unpredictable layouts that can jump as the user shifts his focus. In addition, the implementation of these pretty-print algorithms is tangled with the implementation of the holophrasting technique employed. We have separated the two, reducing the pretty-printer's dependency on the holophrasting technique. This allows our algorithm to flexibly support different holophrasting engines.

Lector [34] is a simple format language based on SGML and a corresponding pretty-printer intended for use by X11 applications. Lector is peephole-efficient, in the sense that it can start at an arbitrary location in the document and layout a documents text from the inside out to fill the current window. However, its format language is less expressive than our LOTs. For example, Lector cannot group line breaks together such that all or none of the line-breaks are taken in a group. Stability of the peephole view is easier to achieve in this more restricted setting. Also, Lector offers fisheye and elision only based on the nested structure of code. Our framework allows fisheyes and elision to be defined using arbitrary functions.

Incremental formatters for LaTeX, like Vortex [25,26], can format LaTeX documents efficiently in a peephole fashion; however LaTeX itself does not allow one to specify how to space optimize the display of structured text for different widths. Anyone who has used the venerable `tabbing` environment to display source code in a draft paper, and then been forced to reformat for a two-column camera-ready format, is painfully familiar with this phenomenon.

Several existing systems support Holophrasting. Jaba [35] is an editor equipped with automated holophrasting and literate programming. Program slicing [36] can also be viewed as a form of holophrasting Furnas' fisheye view [23] is a heuristic technique to automate the folding and unfolding of code. None of these techniques include a space optimizing pretty-printer that can layout text for different display widths.

Syntax-directed editors generally incorporate pretty-printers. There are many excellent systems in this category. We cite [5,11,14,15,37,38] as samples. Some are *incremental* in the sense that after an edit, the refresh does only an amount of work proportional to the size of the edit. When automated refactoring tools are used, these edits may be far reaching, requiring most of the file to be reformatted. Some of these systems use hand-built pretty-printers and suffer from the drawback mentioned in the introduction. Others achieve peephole-efficiency by restricting the possible layouts that can be expressed in their LOT, thereby sacrificing space-optimality. Our peephole algorithm sacrifices neither.

6 Conclusion

We have developed a peephole-efficient, stable version of a space-optimizing pretty-printer. We have also extended the algorithm to incorporate holophrasting techniques and font and font size altering techniques. The *space-optimizing* property (inherited from [13]) allows attractive renderings at different window sizes, without resorting to horizontal scrolling. The *peephole optimization* substantially improves performance. *Stability* allows efficient browsing and scrolling without visually jarring jumps in the display, and without unduly complicating or slowing down the implementation of scrolling and browsing. The hard part of our algorithm is getting peephole-efficiency and stability in the context of a space optimizing pretty-printer; such printers switch layouts to suit the available window width, and make finding anchors tricky.

We have also developed a plug-in for Eclipse that enables peephole pretty-printing while browsing and editing Java source code. Source code, screen shots, and examples for our peephole pretty-printer are available at `http://peephole.cs.ucdavis.edu/` as well as our Eclipse plug-in.

References

[1] B. P. Lientz, E. B. Swanson, G. E. Tompkins, Characteristics of application software maintenance, Communications of the ACM 21 (6).

[2] T. A. Corbi, Program understanding: challenge for the 1990s, IBM Syst. J. (USA) 28 (2), article.

[3] R. M. Bates, A PASCAL prettyprinter with a different purpose, SIGPLAN Notices 18 (3).

[4] R. Baecker, Enhancing program readability and comprehensibility with tools for program visualization, in: ICSE, ACM Press, 1988.

[5] P. Borras, D. Clement, J. Incerpi, G. Kahn, B. Lang, V. Pascual, CENTAUR: the system, SIGPLAN Notices 24 (2).

[6] O. Chitil, Functional Pearl: Pretty Printing with Lazy Dequeues, in: SIGPLAN Haskell Workshop, 2001.

[7] M. O. Jokinen, A language-independent prettyprinter, Software: Practice and Experience 19 (9).

[8] W. Kahl, Beyond Pretty-Printing: Galley Concepts in Document Formatting Combinators, in: G. Gupta (Ed.), PADL Workshop on Practical Aspects of Declarative Languages, LNCS, Springer-Verlag, 1999.

[9] S. D. Swierstra, P. R. Azero, J. Saraiva, Design and implementation of combinator languages (1998).

[10] J. Hughes, The Design of a Pretty-printing Library, in: J. Jeuring, E. Meijer (Eds.), Advanced Functional Programming, Vol. 925, Springer Verlag, 1995.

[11] INRIA, The PPML Manual, INRIA: Centaur Project (1994).

[12] M. van den Brand, E. Visser, Generation of formatters for context-free languages, in: TOSEM, 1996.

[13] P. Wadler, A Prettier Printer, in: The Fun of Programming, Oxford, 2003.

[14] T. Reps, T. Teitelbaum, The Synthesizer Generator: a System for Constructing Language-Based Editors, Springer-Verlag, 1989.

[15] M. de Jonge, A Pretty-Printer for Every Occasion, in: CoSET, 2001.

[16] M. de Jonge, Pretty-printing for software reengineering, in: ICSM, 2002.

[17] R. J. Miara, B. S. Joyce A. Musselman, Juan A. Navarro, Program indentation and comprehensibility, CACM 26 (11).

[18] G. T. Leavens, Prettyprinting styles for various languages, SIGPLAN Notices 19 (2).

[19] G. Blaschek, J. Sametinger, User-adaptable prettyprinting, Software: Practice and Experience 19 (7).

[20] R. C. Waters, User Format Control in a LISP Prettyprinter, TOPLAS 5 (4).

[21] D. C. Oppen, Prettyprinting, TOPLAS 2 (4).

[22] J. Coutaz, The box, a layout abstraction for user interface toolkits, Tech. rep., Carnegie Mellon University, CMU-CS-84-167 (1984).

[23] G. W. Furnas, Generalized fisheye views, in: M. Mantei, P. Orbeton (Eds.), CHI, ACM Press, 1986.

[24] T. Ball, S. G. Eick, Software visualization in the large, Computer (USA) 29 (4), article IEEE Comput. Soc Access restricted.

[25] P. Chen, M. A. Harrison, I. Minakata, Incremental document formatting, in: Proc. of the ACM Conference on Document Processing Systems, ACM Press, 1988.

[26] P. Chen, J. Coker, M. A. Harrison, J. W. McCarrell, S. Proctoer, The vortex document preparation environment. 45-54 b, in: TEX for Scientific Documentation, Vol. 236 of Lecture Notes in Computer Science, Springer, 1986.

[27] D. Nguyen, S. B. Wong, Design patterns for lazy evaluation, in: SIGCSE Technical Symposium on Computer Science Education, ACM Press, 2000.

[28] G. J. Badros, JavaML: A markup language for Java source code, Computer Networks 33 (1–6).

[29] C. S. Collberg, S. Davey, T. A. Proebsting, Language-agnostic program rendering for presentation, debugging and visualization, in: IEEE International Symposium on Visual Languages, 2000.

[30] R. Baecker, A. Marcus, Design principles for the enhanced presentation of computer program source text, in: CHI, 1986.

[31] M. Ruckert, Conservative pretty printing, ACM SIGPLAN Notices 32 (2).

[32] M. Mikelsons, Prettyprinting in an interactive programming environment, SIGPLAN Notices 16 (6).

[33] S. R. Smith, D. T. Barnard, I. A. Macleod, Holophrasted Displays in an Interactive Environment, Int. J. Man-Mach. Stud. (UK) 20 (4), article.

[34] D. R. Raymond, Flexible text display with lector, Computer 25 (8).

[35] A. Cockburn, Support tailorable program visualisation through literate programming and fisheye views, Information and Software Technology 43.

[36] F. Tip, A survey of program slicing techniques, Journal of programming languages 3.

[37] A. Habermann, D. Notkin, Gandalf: Software development environments, IEEE Transactions on Software Engineering.

[38] W. W. Pugh, S. J. Sinofsky, A new language - independent prettyprinting algorithm, Tech. Rep. TR87-808, Cornell University, Computer Science (87).