JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2012; **00**:1–24
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr

Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation

Xiaoran Wang[1], Lori Pollock[1] and K. Vijay-Shanker[1]

[1]*Computer and Information Sciences, University of Delaware, Newark, DE 19716 USA*

SUMMARY

Good programming practice and guidelines suggest that programmers use both vertical and horizontal spacing to visibly delineate between code segments that represent different algorithmic steps or high level actions. Unfortunately, programmers do not always follow these guidelines. Editors and IDEs can easily indent code based on syntax, but they do not currently support automatic blank line insertion, which presents more significant challenges involving the semantics.
This paper presents and evaluates a heuristic solution to the automatic blank line insertion problem, by leveraging both program structure and naming information to identify "meaningful blocks", consecutive statements that logically implement a high level action. Our tool, SEGMENT, takes as input a Java method, and outputs a segmented version that separates meaningful blocks by vertical spacing. We report on several studies involving human judgements to evaluate the effectiveness of the automatic blank line insertion algorithm, for different size methods and for different levels of programmer expertise. The results indicate strong positive overall opinion of SEGMENT's effectiveness in comparison with both developer-written blank lines and blank lines inserted by newcomers to the code. The results vary only slightly among short and long methods, and among novice and advanced programmers. SEGMENT assists in making users obtain an overall picture of a method's actions and comprehend it quicker as well as provides hints for internal documentation placement. Copyright © 2012 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Program understanding; readability; software tool; automatic formatting

1. INTRODUCTION

A major time-consuming activity during software maintenance is reading source code [1–3] to gain understanding adequate to identify potential improvements and make desired modifications. The source code serves as both a human-readable form of the programmer's algorithms and data structures, and the compilable program for execution.

In addition to the code's size and complexity, the readability of source code can be affected by the identifier names, comments, and overall appearance. [4]. This observation led to the invention of prettyprinters [5], which automatically reformat a source code to improve the appearance or fit a specific coding style. As separate tools or within IDEs, these prettyprinters perform textual transformations such as placing certain kinds of statements on separate lines, indenting and left-justifying certain lines, inserting blank lines before specific syntactic units such as certain keywords, or removing extra blank space. These transformations are purely textual, not semantic, and typically use the keywords and maybe the syntax to identify and perform transformations. While code

Contract/grant sponsor: National Science Foundation Grant No. CCF-0702401 and CCF-0915803.

readability involves the syntactic appearance of code, poor readability is perceived as a barrier to program understanding, which focuses on the semantics [6].

Similar to indentation, vertical spacing (i.e., inserting blank lines) is believed to help readability by visibly separating code segments into logically related segments. According to Sun's Java code conventions [7], blank lines improve readability. Software engineering books [8, 9] suggest using plenty of blank lines to break up big blocks of code. Recent studies with humans judging software readability [10] even suggest that simple blank lines are more important than comments to local judgments of readability.

Despite its believed usefulness in readability, vertical spacing is not used as it should in practice. In a study of 16,236 methods that have 16-40 lines of code, 35% methods contain *only 0-1 blank line*. In addition, manual inspection showed that many larger methods have some, but very few blank lines, resulting in large blocks of code with no vertical segmentation for readability. In general, we also found that developers use vertical spacing inconsistently.

Integrated development environments can easily indent code based on syntax, but do not currently support automatic blank line insertion, which presents significant challenges involving the semantics. Automatic blank line insertion requires some notion of what constitutes a logically related code segment and then an algorithm for automatically identifying them such that humans reading the segmented code are helped, and not hindered, in their understanding of the code's actions.

This paper presents a heuristic solution to the automatic blank line insertion problem. We leverage both program structure and naming information to identify "meaningful blocks", consecutive statements that logically implement a high level action. Our tool, SEGMENT, takes as input a Java method, and outputs a segmented version that separates meaningful blocks by blank line insertions. Not only can the resulting segmentation help in readability, it can provide hints for where to place internal comments. SEGMENT is intended for new readers to the code, or developers who are now reading their code after many changes by other developers. Thus, it can be used in different modes of operation based on the reader's preference for seeing the original blank lines only, original plus automatically inserted, or only automatically inserted blank lines.

The main contributions of this paper are:

- Characterizations of kinds of meaningful blocks composed of consecutive statements,
- An algorithm and set of heuristics to automatically segment a large sequence of statements into meaningful blocks, and
- Results from several studies indicate strong positive overall opinion of SEGMENT's effectiveness in comparison with developer-written blank lines and blank lines inserted by newcomers to the code. The results vary only slightly among short and long methods, and among novice and advanced programmers.

This paper significantly extends the initial empirical evaluation in our previous paper to focus on more in-depth questions and larger data sets for investigating the effectiveness of our strategy for blank line insertion [11]. Specifically, we increased the size of our data, investigated several new questions in the evaluation, to gain more understanding of users' views of our output and future improvements needed, and added a detailed analysis of the results, specifically where the system could be improved by future research.

2. THE REVERSE ENGINEERING PROBLEM

To better understand how developers use vertical spacing, we analyzed developers' placement of blank lines in a large corpus of 24 projects of Java programs. We randomly selected methods and manually characterized the common contexts of developers' blank lines, some of which indeed follow Java code conventions [7]. Common uses of vertical spacing that we observed in our corpus include:

- After all local declarations at the top of a method body

DOI: 10.1002/smr

- Before each class instance creation, especially when setting fields after the creation
- Before and after a sequence of very similar looking statements
- Before and after a data flow chain
- Before and after a sequence of setters
- Before and after a try-catch statement
- Before and after a synchronized statement
- Before a nest of loop (while, do, for) or 'if' statements
- Before the preamble of variable assignments or declarations immediately preceding a loop/if condition, when the preamble statement definitions are used in the loop/if condition
- Before a comment block
- Before a return statement

Figure 1 shows an example method that illustrates how blank lines are used. Note how the readability of the code presented in part (a) is improved by inserting blank lines as in part (b). The blank line at line 6 is after an `if` block and before an initialization block. Line 10 follows an initialization block. The blank line at line 12 is before a data flow chain. Line 17 separates two data flow chains. The blank line at line 22 follows a data flow chain.

Since programmers both read and write code, using their code-writing behavior seems a good guide for developing heuristics for automatically inserting blank lines. Using a corpus for learning programmer behavior, one can reverse engineer a sense of the kinds of meaningful blocks that humans visibly segment. Then, characteristics about the statements surrounding the separation points and statements kept in the same block can be used to develop a set of features to suggest that consecutive statements should either be separated or put together. Our analysis of programmers' vertical segmentation of methods suggested that discriminating statement features might include:

- Programming language syntax
- Variables being defined/updated/used within each statement
- Names being used in each statement and how they are being used

Learning the segmentation behavior of programmers is complicated by not knowing the programmers' intent in blank line insertion. While the overall goal is to segment into logically related blocks for increased readability, there could be other factors. For instance, they may have inserted a blank line due to size only, to break up large, but logically related block. Similarly, a given statement may fit in either of a pair of consecutive blocks logically, and the reason for the programmers' choice is not easily identified.

3. AUTOMATIC SEGMENTATION

Our approach to automatically inserting blank lines into method code follows the process shown in Figure 2. The input is the method body statements in the form of an abstract syntax tree, with their associated variable definitions and uses. The first phase identifies initial blocks that our heuristics suggest are logically related blocks. For example, consider the following code snippet.

```
1  setShowGrid(true);
2  setIntercellSpacing(new Dimension(1,1));
3  tree.setRootVisible(false);
4  tree.setShowsRootHandles(true);
5  TreeCellRenderer r = tree.getCellRenderer();
6  r.setOpenIcon(null);
7  r.setClosedIcon(null);
8  r.setLeafIcon(null);
```

The first phase, 'Identify Initial Blocks', identifies three blocks of logically related, consecutive statements:

```
setShowGrid(true);
setIntercellSpacing(new Dimension(1,1));
```


Prepared using smrauth.cls
DOI: 10.1002/smr

```
1  public void runSupport() {
2    SWTSkinObject soSearchResults = getSkinObj("search");
3    if (soSearchResults == null) {
4         return;
5    }
6    Control controlTop = browserSkinObject.getControl();
7    Control controlBottom = soSearchResults.getControl();
8    Browser search = soSearchResults.getBrowser();
9    soSearchResults.setVisible(false);
10   FormData gd = (FormData) controlBottom.getLayoutData();
11   gd.top = null;
12   gd.height = 0;
13   controlBottom.setLayoutData(gd);
14   gd = (FormData) controlTop.getLayoutData();
15   gd.bottom = new FormAttachment(controlBottom, 0);
16   gd.height = SWT.DEFAULT;
17   controlTop.setLayoutData(gd);
18   controlBottom.getParent().layout(true);
19   search.setUrl("about:blank");
20 }
```

(a) *No vertical spacing*

```
1  public void runSupport() {
2    SWTSkinObject soSearchResults = getSkinObj("search");
3    if (soSearchResults == null) {
4         return;
5    }
6
7    Control controlTop = browserSkinObject.getControl();
8    Control controlBottom = soSearchResults.getControl();
9    Browser search = soSearchResults.getBrowser();
10
11   soSearchResults.setVisible(false);
12
13   FormData gd = (FormData) controlBottom.getLayoutData();
14   gd.top = null;
15   gd.height = 0;
16   controlBottom.setLayoutData(gd);
17
18   gd = (FormData) controlTop.getLayoutData();
19   gd.bottom = new FormAttachment(controlBottom, 0);
20   gd.height = SWT.DEFAULT;
21   controlTop.setLayoutData(gd);
22
23   controlBottom.getParent().layout(true);
24   search.setUrl("about:blank");
25 }
```

(b) *With vertical spacing*

Figure 1. Example Uses of Blank Lines for Enhanced Readability

```
tree.setRootVisible(false);
tree.setShowsRootHandles(true);
TreeCellRenderer r = tree.getCellRenderer();
```

```
TreeCellRenderer r = tree.getCellRenderer();
r.setOpenIcon(null);
r.setClosedIcon(null);
r.setLeafIcon(null);
```


DOI: 10.1002/smr

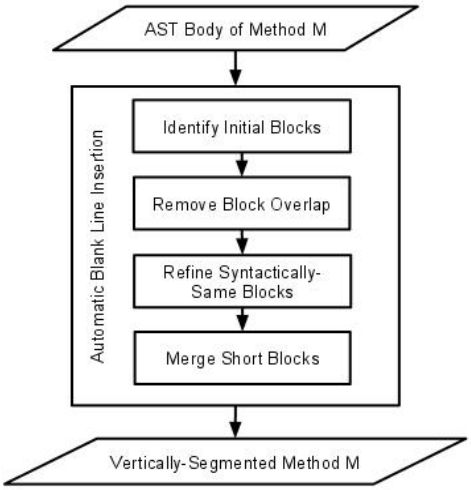


Figure 2. *Automatic Method Segmentation Process*

Because this phase is focused on identifying logically related, consecutive statements without concern for overlap, it sometimes results in statements at the separation points being included in two blocks. In the example, blocks 2 and 3 both include the `Tree CellRenderer ...getCellRenderer` method call statement. We say there is an overlapping statement in overlapping blocks at the point of separation. To achieve method segmentation with no repeated statements, the second phase, 'Remove Block Overlap', decides where to place each statement involved in overlapping blocks, such that each statement in the method is included in exactly one block. Our heuristics determine that the `Tree CellRenderer ...getCellRenderer` method call statement in the example should be placed in the third block.

Sometimes, the output of the first two phases results in some very long blocks, particularly for blocks of a certain kind called syntactically-same blocks. The third phase, 'Refine Syntactically-Same Blocks' further segments these large blocks for more meaningful blocks. At this point, there may be some blocks that contain only a single statement. The final phase determines the potential merge of these small blocks with neighboring blocks.

Each remaining subsection describes the individual phases of our approach. The algorithm is applied first on the overall method, and then is applied recursively on the bodies of compound statements including conditionals, loops, try, synchronized. when the bodies consist of 4 or more statements.

3.1. Phase I: Identify Logically-related Blocks

We define a **meaningful block** to be a sequence of consecutive code lines in a Java method, in which the lines of the block are somehow related meaningfully to represent a single high level concept or action. Humans can quickly recognize many of these concepts or actions. These high level actions or concepts may be expressible as a single unit by abstracting the code sequence in a method, but would be impractical to abstract in that way. Henceforth, we use **block** to mean meaningful block.

Based on our analysis of programmers' use of blank lines for segmenting method bodies, we identified three major kinds of blocks: syntactically-same, data-flow chain, and extended-SWIFT. In addition, each return statement is separately treated as a block. One blank line should always be used before internal comments, according to the Java code convention [7].

Syntactically-Same Blocks (SynS). A syntactically-same block is a consecutive sequence of statements in which every statement belongs to the same *syntactic category*, thus related through syntax. A statement's *syntactic category* is its class of programming language syntax, which we

DOI: 10.1002/smr

enumerate as one of {init, declare, assign, method call, object method call, other}, as exemplified in Table I. In Phase III, a SynS block may be further segmented.

Table I. Syntactic Categories with Examples

Syntactic Category	**Example Format**
init	type name = ...;
declare	type name; type name-1, name-2, ..., name-n;
assign	name = ...; name.property = ...;
method call	methodcall(...);
object method call	object.methodcall(...);
Others: super method call postfix expression prefix expression infix expression throw return	 super(. . .); i–; ++i; a+b; throw . . . ; return ...;

Data-Flow Chain Blocks (DFC). A data-flow chain block is a consecutive sequence of statements that are related through data flow. Based on our analysis of blocks involving data flow, we identified four major kinds of data-flow chain blocks, which vary based on occurrences of variables' initializations, definitions and uses. A variable is said to be *initialized* if it is on the LHS of an *init* statement, or in a *declare* statement. A variable is said to be *defined* if it is on the LHS of an *assign* statement, or it is an object on which a method is invoked. Any variable that appears on the RHS or in a parameter is considered to be a variable *use*. We define and show an example of each of the four kinds of DFC blocks here:

- *Initialize-Access (IA):* A variable is initialized in the first statement and then is defined or used in each of the following statements. The DFC block ends before the first statement that does not define or use the variable. In this example, variable `classes` is initialized and then some part of it is defined in all the statements of the rest of the DFC block.

```
Iterator[] classes = new Iterator[4];
classes[0] = sNode.eIterator("class");
classes[1] = sNode.eIterator("subclass");
classes[2] = sNode.eIterator("joined");
classes[3] = sNode.eIterator("union");
```

- *Define-Use (DU):* A variable is defined in the first statement and used in all of the next statements. Here, the variable `translationGroup` is defined as an object on which the method `addChild` is invoked, and then used as a parameter in a method call in the next statement.

```
translationGroup.addChild(new Sphere());
writeNode(translationGroup);
```

- *Initialize-n-Use-n (InUn):* A sequence of statements initialize variables that are all used by the same statement, which immediately follows the last of the sequence of initializations. The last statement of this DFC block uses both `sound` and `frame` which were immediately initialized by preceding statements.

```
Frame frame = movie.appendFrame();
Sound sound = helper.getSoundDefinition();
int frames = frame.startSound( sound, 30 );
```


DOI: 10.1002/smr

- *Define (DD):* A sequence of consecutive statements that invoke some method on the same object. In this example, all three consecutive statements define `cfw` in some way through a method call.

```
cfw.addLoadThis();
cfw.addALoad(CONTEXT_ARG);
cfw.addALoad(SCOPE_ARG);
```

Extended SWIFT Blocks (E-SWIFT). We refer to the set of compound statements including {switch, while, if, for, try, do, synchronized} as SWIFT statements. An extended-SWIFT block includes a SWIFT nest extended with the immediately preceding lines that initialize or define a variable that is used in the condition controlling the SWIFT statement nest. We call the preceding lines the preamble. In the example below, the SWIFT statement is a simple 'if' statement; the variable `month` is defined in the preamble and used in the condition controlling the SWIFT statement body.

```
month = month % 12;
if (month < 0)
    month += 12;
```

3.2. Phase II: Remove Block Overlap

It is straightforward to separate the logically related blocks identified by the first phase when there are no overlapping statements (i.e., a line that belongs to two consecutive blocks at the same time). However, two consecutive blocks frequently have overlap. The second phase focuses on making decisions on which block to include an overlapping statement such that the code is segmented.

The intuition behind the approach is that DFC and extended-SWIFT blocks are blocks with statements that are logically related due to program structure, either data flow or control flow. We want to keep statements that have been included in those blocks and also in a neighboring block with these kinds of blocks because of the programmatic relation with the rest of the DFC or extended-SWIFT block.

However, it might be the case that the overlapping statement has a strong semantic relation with the neighboring block that outweighs its programmatic relation with the DFC or extended-SWIFT block. Specifically, the neighboring block might be a syntactically-same block in which the overlapping statement has similarities with the rest of the SynS block beyond being in the same syntactic category, in which case, we might want the overlapping statement to be put with the neighboring SynS block instead of the DFC or extended-SWIFT block.

Consider Figure 3, where lines 1 and 2 form an SynS block, and lines 2 and 3 form a DFC block. While the DFC block typically has a strong relation between its statements, we want to keep lines 1 and 2 together because they are not only both initializations (i.e., syntactically-same), but the words in their names indicate they are also very similar semantically. This situation motivates a measure of different levels of similarity between consecutive statements (statement pairs) in a SynS block.

```
1 ActionMap am = tlb.getActionMap();
2 InputMap im = tlb.getInputMap();
3 im.put(prefs.getKey(), "close");
4 am.put("close", closeAction);
```

Figure 3. Overlapping Statement between SynS and DFC Blocks

We define a statement-pair similarity level for consecutive statements of the same syntactic category based on word usage and naming conventions. The statement-pair similarity level is computed differently for each syntactic category due to the available information in that kind of statement. The similarity level is defined as 1, 2, or 3, with 3 being the most similar. Similarity level is defined as 0 if there is no added similarity beyond syntactic category.

The complete set of similarity levels and their corresponding conditions for each syntactic category are shown in Table II. 'name' refers to the identifier name of each syntactic category given

DOI: 10.1002/smr

in Table I , while RHS refers to the right hand side of an assignment or initialization statement. For example, as indicated by the row labeled "object-method call", the similarity level between a pair of object method call statements is defined as 1 if either the object or method names are similar to each other, while it is defined as level 2 if both object and method names are similar, respectively.

Table II. Statement-pair Similarity Levels in a SynS Block

Syntactic type	**Levels of Similarity**		
	1	**2**	**3**
init	type or RHS	type + name or type + RHS	type + name + RHS
declare	type	-	-
assign	name or RHS	name + RHS	
object-method call	object or methodname	object + methodname	
method call	methodname	-	-

The statement-pair similarity level computation utilizes notions of id (name) similarity and RHS similarity. We now describe our *ID-Similarity* function and *RHS-Similarity* function and then finally explain how we make decisions on which block to include the overlapping statements, based on the similarity levels in neighboring SynS blocks.

3.2.1. ID-Similarity Function The ID-Similarity function returns true or false based on the similarity in appearance of two variable names, type names or method names given as input. It is called to implement the statement-pair similarity table, whenever similarity between two types, variable names, object names, or method names is needed. For the purpose of blank-line insertion, similarity of two identifiers is defined in terms of appearance, not based on meaning.

The first step is to split the input strings by camel case letters (i.e., capitalized substrings), special characters such as underscore, and numbers. We call each of the component substrings of each identifier, words, while there may be abbreviations or non-dictionary words. We then apply our heuristic for ID-Similarity with the purpose of identifying consecutive statements that are very similar semantically. One might consider different heuristics to define ID-Similarity. Our heuristic is based on our observations of where blank lines were inserted by humans over many examples.

Our heuristic separates the determination of ID-Similarity into several cases. If both identifiers are single words, ID-Similarity returns true. For example, ID-Similarity is true for names `x` and `y`. If the identifiers are multi-word of the same number of words, and there is at least one exact matching word in the same position of the identifiers, ID-Similarity returns true. For example, ID-Similarity is true for `strTmp` and `strMsg`. If the identifiers are multi-word with one identifier having one more word than the other and either the first or last words match, ID-Similarity returns true. For example, ID-Similarity is true for `srcPixels` and `dstInPixels`. ID-Similarity returns false for all other situations.

3.2.2. RHS-Similarity Function The RHS-Similarity function takes two RHS's as input and returns true or false based on a simple similarity measure. The RHS of an assignment or initialization statement can be any of: constant, single variable name, class instance creation, infix expression, prefix expression, postfix expression, cast expression, method call, or object method call.

If both RHSs are the same syntactic category (e.g., both class instance creations), then RHS-Similarity returns true. If both are either method calls or object method calls, then ID-Similarity is called on RHSs. If ID-Similarity returns true, then RHS-Similarity returns true, else false. This is illustrated below:

Input: *getPrintAction();* and *getCloseAction();*
RHS-Similarity returns: *true*

DOI: 10.1002/smr

```
1 Tree t = new Tree();
2 t.getNodeTable().addColumns(LABEL_SCHEMA);
3
4 Node r = t.addRoot();
5 r.setString(LABEL, "0,0");
```

Figure 4. Example (DFC DFC) Consecutive Blocks; Line 4 Overlap

3.2.3. Deciding Block for Overlapping Statement The placement of an overlapping statement among two consecutive blocks depends on the kind of each consecutive block, among SynS, DFC, and Extended SWIFT. Overlapping statements may occur between all block sequences except any blocks following an Extended SWIFT block. Here, we describe the heuristics for placing the overlapping statement into the appropriate block of a consecutive block pair.

(DFC, DFC) Pair. If the overlapping statement S is an *init* statement, then S is placed as the first statement of the second DFC block. Otherwise, if S is any other kind of statement, the two DFC blocks are merged into a single DFC block. For example, in Figure 4, Line 4 is the overlapping statement, which was included in the first DFC because of the use of `t` and included in the second DFC block due to the definition of `r`. We would segment before line 4, and place it in the second DFC.

(SynS, DFC) / (SynS, E-SWIFT) / (DFC, SynS) Pairs. Normally, the DFC and Extended SWIFT relations between statements is stronger than SynS relations, and the overlapping statement would be placed in the DFC or Extended SWIFT block. For example, in Figure 5., Line 3 is placed in the DFC block, although it is also involved in the preceding SynS block initially.

```
1 Image image = ii.getImage();
2
3 MediaTracker mt = new MediaTracker();
4 mt.addImage(image, 0);
5 mt.waitForID(0);
```

Figure 5. Example (SynS DFC) Consecutive Blocks; Line 3 Overlap

Similarly, in Figure 6, Line 3 is put in the Extended SWIFT block, despite also being initially part of the SynS block due to its similar syntactic category to line 1.

```
1 year = Math.floor(month / 12);
2
3 month = month % 12;
4 if (month < 0)
5             month += 12;
```

Figure 6. Example (SynS E-SWIFT) Consecutive Blocks; Line 3 Overlap

To determine the exceptional situations when the overlapping statement should be placed with the neighboring SynS block instead of the DFC or Extended SWIFT block, we use the statement-pair similarity level. We say that two consecutive statements are *highly similar* when the statement-pair similarity level is 2 or 3, for those syntactic categories that have defined levels at least 2. For statements in the method call or declare syntactic categories, we designate level 1 similarity as *highly similar*. In Figure 7, lines 2 and 4 form a DFC block, but Line 4 and 5 form a *highly similar* SynS block. Thus, Line 4 is placed in the SynS block.

```
1 icon_url = getResource();
2 icon = new ImageIcon(icon_url);
3
4 putValue(Action.SMALL_ICON, icon);
5 putValue(Action.SHORT_DESCRIPTION, description);
```

Figure 7. Placing Overlapping Statements using Highly Similar Statements

(DFC, E-SWIFT) Pair. We performed a manual analysis that involved examining the properties of blocks between developer inserted blank lines and the frequencies that these properties occurred.

DOI: 10.1002/smr

Based on our manual analysis of blank line usage, we consider the DFC relation to be stronger than E-SWIFT, thus the overlapping statement is placed in the DFC block. In Figure 8, Line 3 is placed in the DFC block, despite initializing a variable that is used in the SWIFT condition.

```
1 int imageWidth = image.getWidth(null);
2 int imageHeight = image.getHeight(null);
3 int imageRatio = imageWidth / imageHeight;
4
5 if (thumbRatio < imageRatio) {
6     thumbHeight = (thumbWidth / imageRatio);
7 } else {
8     thumbWidth = (thumbHeight * imageRatio);
9 }
```

Figure 8. Example (DFC E-SWIFT) Consecutive Blocks; Line 3 Overlap

3.3. Phase III: Refine Syntactically-Same Blocks

This phase segments large SynS blocks that sometimes result from the initial block construction with the goal of providing more readability. These blocks will sometimes contain subsequences of consecutive statements that are more similar than others. In Figure 9, Lines 1-4 are more similar to each other than they are with Line 6.

```
1 im.put(prefs.getKey("Close entry"), "close");
2 am.put("close", closeAction);
3 im.put(prefs.getKey("Print entry"), "print");
4 am.put("print", printAction);
5
6 tlb.setFloatable(false);
```

Figure 9. Candidate Syntactically-Same Block for Refining

The decision to further segment a SynS block depends on (a) the length of the block and (b) whether there exist consecutive subsequences with different similarity levels. If a block only contains 2 or 3 statements, there is no need to segment it further. However, if it contains more statements, it is reasonable to segment them. If a sequence contains statements that are all the same similarity to each other, there is no need to break them. However, readability may be improved by segmenting into groupings that have different similarity levels.

To cluster those more similar statements together and segment them from others, we designed an algorithm to cluster syntactically-same statements based on statement-pair similarity level. The clustering algorithm performs a sequential scan through the SynS block with a sliding window of three statements, a, b, c. At each point in the scan, the similarity levels of each of the two pairs is computed, and the difference between the similarity levels of the two statement pairs involved in the three statements (a, b) and (b, c) is computed. If this difference in levels is nonzero, then we insert a blank line between the pair with the lowest similarity level. At the end of the algorithm execution, there will be a blank line between groupings of consecutive statements with the same similarity level.

Consider the example shown in Figure 10. Lines 1,2, 3 and 5-6 have similarity level 1; lines 3-5 have similarity level 0. The difference in similarity levels is 1 between pairs (2,3) with similarity level = 1 and (5,6) with similarity level 0. Since (2,3) have a higher similarity level, a blank line is inserted at line 4 with the lower similarity level between the statement pair (3,5).

```
1 in.start();
2 out.start();
3 error.start();
4
5 out.join();
6 error.join();
```

Figure 10. Example of SynS Statement Clustering

DOI: 10.1002/smr

3.4. Phase IV: Merge Short Blocks

Sometimes because we examine statement pairs individually, we create very small blocks of 1-3 statements, which may harm readability. This last phase focuses on merging very small blocks.

We currently target single-line blocks, with the goal of merging them with one of their neighboring blocks. They could result from not being included in a SynS, DFC, or Extended SWIFT block in phase 1, or they could have been created through the segmenting of SynS blocks. Figure 11 shows an example of three statements that were single-statement blocks after the first phases, but can be merged into a single block because they have similar RHSs. Our strategy is to look for similar features between the single-line block and its neighboring blocks using features summarized in Table III. For example, a single-line block that is an init statement neighboring a single-line block that is a declare statement would be merged if their types are similar. An init statement neighboring an assignment statement would be merged if the right-hand-sides are similar or names of the left hand side variable are similar.

```
JLabel label1 = new JLabel("Search Name:");
searchNameField = new JTextField(12);
JLabel label2 = new JLabel("Search Type:");
```

Figure 11. Example Single-line Blocks Being Merged

Table III. Similar Features between different kinds of statements

Syntactic Categories	Example Format	Condition
init & declare	type name = ...; type name;	type similar
init & assign	type name = ...; name = ...;	RHS or name similar
methodcall & object-methodcall	methodcall(parameter); object.methodcall(parameter);	method call or param similar

3.5. Different Modes of Operation

SEGMENT could be used in different modes of operation, including: (1) start from no blank lines and insert all blank lines automatically using our heuristic constructed toward new readers, (2) start with existing blank lines and insert additional blank lines, but do not eliminate the original blank lines, or (3) work in mode 2 and color code the original and automatically inserted blank lines differently. The tool can work as only a display mode tool or a tool that actually changes the existing files. The evaluation is performed with discarding the original blank lines to determine whether the automatic tool can produce vertically spaced code that is effective from a reader perspective. Thus, the original developers or new readers have options to view the code in different ways based on their preference.

4. EVALUATION

4.1. Overview

We implemented the automatic blank line insertion algorithm described in Section 3 as a prototype tool called SEGMENT, for Java methods. In this section, we describe our evaluation of the effectiveness of SEGMENT. Specifically, our evaluation focuses on the following questions. Does SEGMENT:

- insert enough blank lines?
- insert too many blank lines?

DOI: 10.1002/smr

- insert at appropriate program locations?

In addition, we also explore the following questions:

- How does method size affect the effectiveness of our strategy? Does our strategy work as well for short methods as long methods?
- How does the user expertise level affect the effectiveness of our strategy? Do novices to programming and advanced software developers expect blank lines to be located in the same locations or different locations? Do they view the blocks in the same ways?

We first describe the sets of human subjects, the selection of methods, and our preliminary study into the subjectivity of blank line insertion. We then present the methodology and results for: (1) A study in which blank lines are inserted by people (newcomers to the code) reading the method code and compared against SEGMENT-inserted blank lines. (Code Readers' perspective vs SEGMENT), (2) A study comparing SEGMENT-generated blank lines with original developers' usage of blank lines. Developers usage should reflect how developers conceptualize code as different blocks. Their insertions inform us on how code is segmented into logical blocks from a developer's (writer's) perspective. (3) A study of human opinions comparing the differences between developer-inserted and SEGMENT-inserted blank lines. Since we randomly selected our evaluation data set from open source projects, the data set could include methods where developers may not have paid sufficient attention to blocks and blank line insertions. In addition, we want to know how readers of the code view SEGMENT-inserted blank lines versus those inserted by humans. (4) A study comparing the effectiveness of SEGMENT for short versus long methods. (5) A study comparing the effectiveness of SEGMENT according to novices versus advanced software engineers. Finally, we perform an analysis of SEGMENT's errors to provide insight into future enhancements.

For all the studies, we engaged 15 advanced-level programmers as our human subjects, including software engineers and graduate students. These advanced programmers have programming experience with C++/Java ranging from 4 to 20 years, with a median of 7 years. All studies except the last study evaluating SEGMENT's effectiveness for different level programmers were done by the advanced programmers. For the study comparing the human-inserted blank lines against SEGMENT's insertions, the annotators inserted the blank lines. The comparison between human-inserted blank lines and SEGMENT's insertions was done manually by the authors in an automated manner. For the study comparing SEGMENT insertions versus developers, the same human subjects were used; however, they only played the role of evaluator as there was no human annotation.

For all studies, we chose methods randomly (except for an effort to keep an even balance between short and long methods) from seven projects (from 18,186 methods). Table IV provides information about the non-trivial-sized, open-source projects we used from sourceforge [12]. Although we are studying blank lines, evaluators need to read the entire method to judge blank line insertion. Thus, we selected methods ranging in size from 10 to 50 lines, neither too short for blank line segmentation nor too long to make the task too tedious. We chose 50 instead of a smaller method length so we could also examine how shorter and longer methods affect the results, thus so we also separated the range into two subranges. We define methods of length 10-24 lines of code to be *short* methods and methods 25-50 lines of code to be *long* methods. We separated the results for both the study on code reader-inserted vs SEGMENT and original developer-written vs SEGMENT by short and long methods and performed separate analyses for comparison. The distribution of short and long methods was kept the same for each set of 100 methods, namely, 50 short and 50 long methods. The third column of the table shows the number of methods in each project ranging between 10 and 50 lines of code. Figure 12 depicts a histogram of the total number of methods over all projects of each size between 10 and 50 lines of code.

Most studies involve human subjects. At the beginning, we conducted a preliminary study to better understand the subjectivity of the blank line insertion task and how humans may differ in their opinions of where blank lines should be used. We gave 3 methods with no blank lines to 12 (advanced programmer) annotators and asked them to insert blank lines at appropriate places. We found that at least 2 out of 3 agreed on nearly every blank line location. Based on these results, we

DOI: 10.1002/smr

Table IV. Subject projects in study. kloc: 1000 lines of code

Project	#methods	# 10-50 loc methods	kloc
GanttProject	4956	755	60
Jajuk	2139	621	44
PlanetaMessenger	1142	357	22
Dguitar	1211	338	20
DrawSWF	2747	620	41
JRobin	1913	556	30
SweetHome3D	4078	1129	73

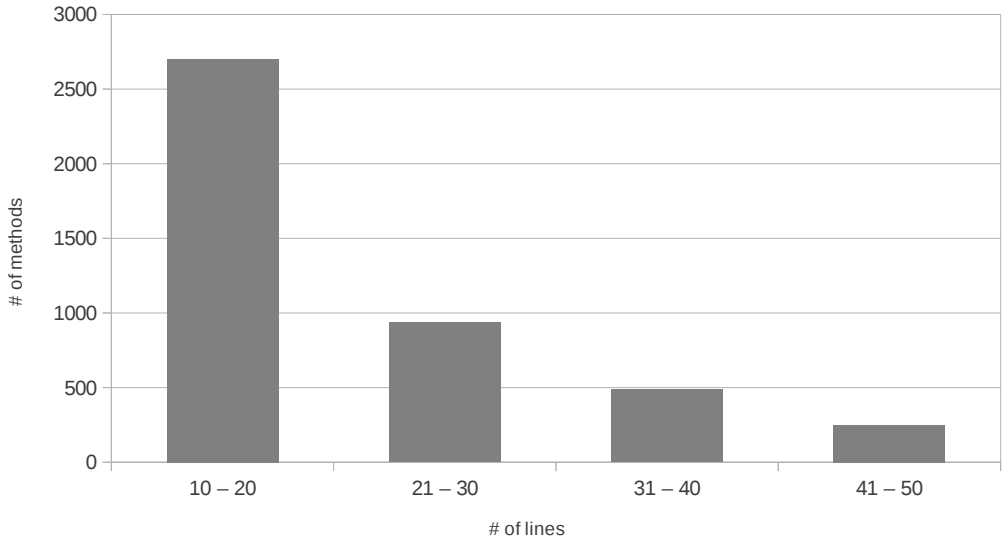


Figure 12. Histogram of method size distribution

developed our studies to have each method be randomly assigned to 3 human subjects and take the majority opinion when there was no unanimous opinion.

4.2. Code Reader's Perspective versus SEGMENT-inserted

The first study was designed to learn how well SEGMENT performs versus what readers new to a code would expect to help them read the code. Unlike developers who may or may not be concerned with readability and whether the code may need to be understood and maintained by others, these human annotators were instructed to specifically consider the blank line insertion issue in code they had not written or seen prior to the study.

Procedure. To obtain a gold standard for blank line insertions based on a reader's perspective, we gave 15 advanced programmer annotators a set of 20 Java methods without any blank lines and asked them to insert blank lines at appropriate locations. To account for variation in human opinion, we gathered annotations from 3 separate advanced programmers for each of 100 unique methods. Thus, the humans performed 300 method annotations in total. To control for any learning effects, each annotator's 20 methods were shuffled and shown in different orders. Also, the 15 annotators (advanced programmers) were randomly put into 5 groups so we could obtain 3 separate opinions on each method. For each method, we manually compared all 3 human-inserted blank line copies of a given method line by line. We defined the gold standard to be the locations where at least 2 out of 3 evaluators (i.e., a majority) inserted a blank line.

DOI: 10.1002/smr

Results and Discussion. Table V presents the results comparing the locations of blank lines inserted by the majority of human annotators versus inserted by SEGMENT. SEGMENT inserted a total of 523 blank lines over the 100 methods. Of those 523 locations, the majority of the human annotators (either 3 of 3 or 2 of 3 annotators) inserted blank lines at the exact same location 345 times. Even in the remaining 178 locations where SEGMENT inserted a blank line and the majority of human annotators did not, at least one of the human annotators indeed chose to insert a blank line at that point. Also, there were 49 locations where the majority of human annotators inserted a blank line and SEGMENT did not. Among these cases, SEGMENT inserted a blank line within one line above or below the human-inserted blank line location in 6 cases.

Precision represents the percentage of locations that SEGMENT inserted in the correct locations when compared against human annotators reading the code and inserting blank lines. Recall measures SEGMENT's coverage of blank lines inserted by newcomers reading the code. From Table V, we can see that recall is high (87.6%), but the precision (66%) is lower than recall. That suggests that SEGMENT is good at inserting blank lines in the correct locations, but appears to be over-inserting blank lines according to newcomers to the code.

Table V. Results of Majority Opinion of Code Readers vs SEGMENT

	Count
Both SEGMENT and Majority Humans (3/3 or 2/3) insert same location (True Positive)	345
Locations SEGMENT inserts and Majority Humans do not (False Positive)	178
Locations Majority Humans insert and SEGMENT does not (False Negative) (Note: When both insert within 1 line of each other, we count as False Negative)	49
Precision (TP/(FP+TP)) = 345/523	66%
Recall (TP/(FN+TP) = 345/394	87.6%

Effectiveness and Method Length. Table VI presents the results comparing the locations of blank lines inserted by the majority of human annotators versus inserted by SEGMENT, separated by short vs long methods. SEGMENT inserted a total of 204 blank lines in the 50 short methods and 319 blank lines in the 50 long methods. Of the 204 inserted into the short methods, the majority of the human annotators inserted blank lines at the exact same location 139 times, resulting in a precision of 68.1%, whereas the precision for long methods is 64.6% with annotators inserting in the same location 206 times of the 319 blank lines inserted by SEGMENT. Thus, the precision for short and long methods are similar, with precision a bit higher for short methods.

Table VI. Results of Majority Opinion of Code Readers vs SEGMENT: Short vs Long Methods

	Short	**Long**
Both SEGMENT and Majority Humans insert same location (True Pos)	139	206
Locations SEGMENT inserts and Majority Humans do not (False Pos)	65	113
Locations Majority Humans insert and SEGMENT does not (False Neg) (Note: When both insert within 1 line of each other, we count as False Neg	22	27
Precision (TP/(FP+TP)) = Short: 139/204; Long: 206/319	68.1%	64.6%
Recall (TP/(FN+TP) = Short: 139/161; Long: 206/233	86.3%	88.4%

For both short and long methods, where SEGMENT inserted a blank line and the majority of human annotators did not (65 cases for short methods and 113 cases for long methods), at least one human annotator matched the SEGMENT-inserted blank line in all cases. The recall for long methods (88.4%) is slightly better than recall for short methods (86.3%), but they are also quite close to each other. In summary, precision and recall are very similar between short and long methods, with slightly better precision for short methods and better recall for long methods, when comparing SEGMENT against what code readers prefer.

DOI: 10.1002/smr

4.3. Developer-written versus SEGMENT-inserted

This study examined how the existing developer-written methods with blank lines compare to methods where the blank lines are completely automatically inserted. The comparison is completely quantitative with no human opinion. The next study examines the human opinions of the differences in these blank line insertions.

Procedure. We chose another set of 100 methods randomly from the same seven projects, such that the methods contained at least one developer-written blank line. We created a copy of each of the 100 methods with all the developer-written blank lines removed. These 100 methods containing no blank lines were input into SEGMENT to produce a SEGMENT version of the same methods.

Results and Discussion. Table VII presents the results comparing the developer-written and SEGMENT-inserted blank line locations. SEGMENT inserted a total of 485 blank lines over the 100 methods. Of those 485 locations, the original developer inserted blank lines at the exact same location 278 times, giving a precision of 57.3%. Of the 75 locations where the original developer inserted a blank line and SEGMENT did not, SEGMENT inserted a blank line within one line above or below the human-inserted blank line location in 21 cases. Recall is 78.8%.

Table VII. Results of Developer-written vs SEGMENT-inserted blank lines

	Count
Both SEGMENT and Developer insert same location (True Positive)	278
Locations SEGMENT inserts and Developer does not write (False Positive)	207
Locations Developer-written and SEGMENT does not insert (False Negative) (Note: When both insert within 1 line of each other, we count as False Negative)	75
Precision (TP/(FP+TP)) = 278/485	57.3%
Recall (TP/(FN+TP) = 278/353	78.8%

Comparing SEGMENT's effectiveness between code readers and code developers, we see that the precision has decreased from 66% against code readers to 57% against the original developers. Recall drops from 87.6% against code readers to 78.8% against the original developers. Thus, SEGMENT's effectiveness overall is more in line with code readers than the original developers. This is not surprising given that common belief is that developers are not always careful about documenting and formatting their codes. They are typically more concerned about developing correct code, rather than worrying about others who might have to read and maintain their code [7, 13–16]. In order to investigate further into these results, we developed a third study to examine whether the results are due to questionable insertions or lack of insertions.

Effectiveness and Method Length. Table VIII presents the results comparing the developer-written and SEGMENT-inserted blank line locations separated by short and long methods. Of the 199 and 286 SEGMENT-inserted blank lines into short and long methods, respectively, the original developer inserted blank lines at the exact same location 118 and 160 times, respectively, giving precision of 59% for short methods and 56% for long methods. Thus, precision is slightly higher for short methods, while recall is higher for long methods, consistent with the comparison of SEGMENT against the code annotators (code readers).

Table VIII. Results of Developer-written vs SEGMENT-inserted blank lines: Short vs Long Methods

	Short	**Long**
Both SEGMENT and Developer insert same location (True Pos)	118	160
Locations SEGMENT inserts and Developer does not (False Pos)	81	126
Locations Developer-written and SEGMENT does not insert (False Neg) (Note: When both insert within 1 line of each other, we count as False Neg	38	37
Precision (TP/(FP+TP)) = Short: 118/199; Long: 160/286	59%	56%
Recall (TP/(FN+TP) = Short: 118/156; Long: 160/197	75.6%	87%

DOI: 10.1002/smr

4.4. Human Opinion of Developer-written vs SEGMENT-inserted

The goal of the third study was to obtain code reader's opinion of the differences between the original developer insertions and SEGMENT's insertions of blank lines, to see which they preferred.

Procedure. Each of the 15 advanced programmers now served as an evaluator of developer-inserted and SEGMENT-inserted blank lines. They were each presented with two copies of 20 methods, for a total of 100 methods (the same methods from section 4.3) and 300 opinions. The first copy contained SEGMENT-inserted blank lines and second copy contained developer-written blank lines. To let evaluators easily compare SEGMENT and developer-inserted blank lines, a web-based evaluation system was developed to show each pair of method copies, left and right on the screen with highlighting by using SyntaxHighlighter [17]. To reduce possible bias, our web system randomly places the SEGMENT-inserted and developer-inserted copies on the left or right part of the screen, and randomly orders each evaluator's 20 methods to avoid learning effect.

We first asked the human evaluators to judge the overall blank line insertion for entire methods. Then, we asked to evaluate each blank line insertion location. When a blank line was inserted by both SEGMENT and developer in the same location, we asked if they agreed with the blank line. When there was a difference between SEGMENT and developers, we asked evaluators which blank line insertion is better or did it not matter for the readability of the code. We also asked overall which method copy with blank lines was better in their opinion.

Results and Discussion. SEGMENT-inserted blank lines were judged to be as good as or better than the original developers by majority opinion for 76% of the 100 methods. The evaluators judged the original developers' blank line segmentation as better overall for only 24% of the methods.

The human evaluators agreed with 271 of the 278 locations(97.5%) where SEGMENT and the original developers inserted blank lines in the same locations. This confirms that code readers highly agree with the locations chosen by both the automatic system and the original developers. In all of the 282 locations where SEGMENT differed in some way from the original developer, the human evaluators preferred SEGMENT's insertion point in 209 cases (74% of the cases). Thus, over all 560 locations analyzed, the human evaluators judged SEGMENT's output as good as or better than the original developer in 209+278 (87%) cases. These results appear to suggest that SEGMENT's insertions are closer to code readers and/or that the developers were not sufficiently careful with formatting the code.

Effectiveness and Method Length. SEGMENT-inserted blank lines were judged to be as good as or better than the original developers by majority opinion for 86% of the short methods and 66% of the long methods. Furthermore, the majority opinion judged SEGMENT as actually better than the developer-written blank lines in 36 of the 50 short methods (76%) and 28 of the 50 long methods (56%).

The human evaluators agreed with 117 of the 118 locations (99%) in short methods and 154 of 160 (96%) locations in long methods where SEGMENT and the original developers inserted blank lines in the same locations. Over all the 237 locations in small methods and 160 locations in long methods analyzed, respectively, the human evaluators judged SEGMENT's output as good as or better than the original developer in 99+118 (91.6%) cases in small methods and 110+160 (83.6%) cases in long methods.

4.5. SEGMENT Effectiveness and Programmer Experience

The goal of this study is to determine if and how the effectiveness of SEGMENT differs from the perspective of novice and advanced programmers. If SEGMENT's effectiveness is judged very differently, we may need to create different versions directed toward the novice and advanced programmers.

Recall that the 15 advanced programmers in this study have a median of 7 years of programming experience in C++/Java. For novice programmers, we recruited 39 volunteer undergraduate students from the beginning of the semester of two sophomore level programming courses. These students

DOI: 10.1002/smr

have completed one Java programming course and are starting their second course involving Java programming.

Procedure. In this study, the novice programmers and advanced programmers examined the same set of 50 randomly selected methods from our initial 100 methods, with the task of judging the original developer-written blank lines and SEGMENT's generated blank lines using the same procedure as described for comparing developer-written and SEGMENT-inserted. The 50 methods are mixed short and long methods and do not have any special distribution requirements, just generally from 15 to 50 lines of code. Each evaluator was randomly assigned 4 pairs of methods which are shown randomly left and right on the screen.

Results and Discussion. For methods overall, the novice majority opinion judged SEGMENT-inserted blank lines to be as good as or better than the original developer in 41 of 50 methods (82%) compared to advanced programmers' majority opinion of 40 of 50 methods (80%). Both the majority opinion of novice and advanced programmers agreed with the 128 locations where both SEGMENT and the original developer inserted blank lines in the same locations. Over all the 119 locations where SEGMENT differed in some way from the original developer, the novices preferred the SEGMENT output in 92 cases (77.3%) while the advanced programmers preferred SEGMENT output in 95 cases (79.8%). In fact, in 29 of 50 methods (58%) , the novice majority opinion favored SEGMENT, and in 33 of 50 methods (66%), the advanced programmers favored SEGMENT over the original developers. These results suggest that a single automatic blank line insertion system suffices for both levels of programmers.

4.6. Analysis of Precision and Recall Errors

4.6.1. Human Opinion of Developer-written better than SEGMENT-inserted. For the 100 methods where the humans judged developer-written versus SEGMENT-inserted blank lines, we analyzed the cases where the human evaluators judged the developer insertion/lack of insertion of blank lines better than SEGMENT.

Precision. We first examined the 49 precision errors, where the human evaluators favored the developer's lack of blank line to SEGMENT inserting a blank line. We discovered several dominant categories, which we call between declare and init statements, merging single-line blocks, and between SWIFT and its preamble.

Between declare and init statements.

There are seven cases where SEGMENT inserted a blank line between `declare` and `init` statements and the original developer did not. In the first example below, SEGMENT inserted a blank line to separate the simple declaration from the following set of declarations with init statements, and in the second example, SEGMENT separated an initialization from later simple declarations without initializations.

```
      int i;
//SEGMENT-inserted blank line
      int aspect_width = 4;
      int aspect_height = 3;
```

```
      NodeList stop_nodes = xml_gradient.getElementsByTagName("stop");
      Color[] colors = new Color[stop_nodes.getLength()];
//SEGMENT-inserted blank line
      Node stop_node;
      NamedNodeMap att_nodes;
```

In our set of 100 methods, there are 11 situations of declare-init which fit one of these cases - declaration without init followed by init, or init followed by declaration without init. In examining the cases all these cases and the human opinion, there is no obvious new rule to apply to improve the precision according to human evaluators. More data is needed to develop a new rule.

Merging single-line blocks.

DOI: 10.1002/smr

Recall from Section 3.4 that SEGMENT only merges single line blocks under certain conditions. There are 11 cases where there are single-line blocks and these conditions are not met and thus SEGMENT does not merge the single-line blocks. It appears that the human evaluators did not care whether these conditions were met, and instead favored merging single-line blocks with no restrictions on when. Thus, the rule could be changed to always merge single-line blocks when they occur. It is interesting to note that all 11 cases appeared at the beginning/end of a method or SWIFT block, never in the middle.

Between SWIFT and its preamble.

Among the 100 methods analyzed, there are 20 cases where SEGMENT applies its rules to insert a blank line between a statement assigning a value to a variable and a SWIFT statement when the variable that is assigned a value does not appear in the SWIFT's condition. In only two of these cases, the human judges preferred to keep the preamble with the SWIFT statement. One case is shown here, which shares the property with the other case that a value (dim, in this case) is assigned a value just prior to the if statement and then reassigned a non-default value within the if statement. Thus, the preamble is an initialization of a default value which is changed depending on a condition, but not used in the condition. These kinds of blocks could easily be recognized and treated as a whole block. All other 18 cases for treated correctly by SEGMENT inserting a blank line according to the human judges.

```
dim = new Dimension(250,50) ;
if(maxTracks >= 0) {
   dim.height = 2*TM + H*(maxTracks+1) + 2 ;
}
```

Recall.

We next examined the 24 recall errors, where SEGMENT did not insert a blank line, the original developer had inserted a blank line, and the human evaluators preferred the developer's blank line.

We identified one major category, which we believe is related to statement length. The developer's insertion of a blank line following a long statement appears to be preferred. The number of characters in a line was not considered during SEGMENT design.

There are nine cases in this category. In 5 of these cases, developers put a blank line after the method signature or a SWIFT condition when they were very long. In the remaining four cases, the developer inserted a blank line after a long statement involving multiple method calls.

```
public JThreadServerUDP( int nInitialPort, int nEndPort ) throws java.io.
    IOException {

 boolean                 bSocketOk = true;
 java.io.IOException     e;
```

```
if( ( strPwd.compareTo( "" ) != 0 ) && ( strConfPwd.compareTo( "" ) != 0 ) ) {

   strUserId.insert( 0, userIdText.getText() );
```

There are 3 cases where the developer inserted a blank line at the end of If/While/Try block and SEGMENT inserts no blank line. For example,

```
do{
//...
  if( nEndIndex == -1 )
     nEndIndex = strInput.indexOf( chEndOfList, nBeginIndex );
    //developer-inserted blank line
 } while( nEndIndex > 0 );
```

```
    if{
    //...
      myProject.getArea().setPreviousStateTasks(null);
           this.setVisible(false);
           dispose();
//developer-inserted blank line
```


DOI: 10.1002/smr

```
        } else if (button.getName().equals("cancel")) {
            this.setVisible(false);
            dispose();
```

```
    try{
    //...
      if( !JKeyParser.parseKeys( inReader, hash, isSpaceSeparator ) ) {
        System.err.println( "JIOStream() - recv(3) - parser error" );
        return false;
      }
//developer-inserted blank line
    } catch( IOException e ) {
      System.err.println( "JIOStream.recv(3) - " + e );
      return false;
    }
```

We examined all similar situations and found that there are 8 else if, 15 try catch, 2 dowhile() blocks where the developer did not insert a blank line consistently. Thus, their insertions seem a bit arbitrary based on the number of such situations and human evaluator agreement with no blank. This does not suggest any new rule.

4.6.2. Differences Between Human Annotators (Code Readers) and SEGMENT. We examined both the precision and recall for the cases where the human annotator blank line insertion differed from SEGMENT.

Precision. There are 75 blank lines inserted by SEGMENT, where no human annotator inserted a blank line.

Ten cases occur where SEGMENT inserted a blank line between 2 large SynS blocks. For example, SEGMENT separated the following large SynS block, but the majority of human annotators did not.

```
    String aux;

    int fret;
    int x;
    int pos;

    Color noteColor;
    Color prevColor;

    GP4Dynamic dynamic;
```

Thirteen cases indicate that the merging in Phase IV is not aggressive enough. Recall Phase IV merges any two or more neighboring single-line blocks where there is at least some similarity between the statements. The example below has two statements that have no similarity, but human annotators kept them together.

```
flags |= 0x2005;

startTag( TAG_DEFINETEXTFIELD, fieldId, true );
```

The third major type of precision error is due to the fact that SEGMENT makes its decisions to insert blank lines oblivious of location in the code. SEGMENT inserted blank lines in seven locations after '}' where it appears there is already sufficient vertical spacing and an additional blank line was not necessary.

Recall.

In the nine cases where all 3 human annotators inserted a blank line and SEGMENT did not, 6 of these locations share the characteristic described in the developer-written vs SEGMENT-inserted study, related to long statements or a statement that has multiple lines.

For example,

```
final HumanResource people = ((HumanResourceManager) getHumanResourceManager())
```


DOI: 10.1002/smr

```
            .newHumanResource();
//human annotator inserted blank line
people.setRole(getRoleManager().getDefaultRole());
```

4.7. Summary of Results

Our study of SEGMENT's effectiveness from a code reader's perspective showed 66% precision and 87.6% recall. This suggests SEGMENT is good at inserting blank lines in the correct locations, but appears to be over-inserting blank lines according to newcomers reading the code. Results from comparing SEGMENT output to the original developers use of blank of lines are lower precision, 57%, and recall, 78.7%. This could be due to developers' focus on correctness with less care about formatting for readability by others. The results show more that SEGMENT is not aggressive enough about merging syntactically similar blocks. This could also indicate that a more aggressive similarity heuristic could be developed. However, that would have to be evaluated in other situations where it currently works fine.

When asked to compare the original developer and SEGMENT blank line insertions, the human judges agreed with 97.5% of the locations where both systems inserted in the same location, and favored SEGMENT's inserting points 74% of the time when the developer and SEGMENT differed. We noticed that some positions of blank lines are subjective. There is no right or wrong answer for every blank line position. For those cases, evaluation shows that our approach is as good as the original developers.

We found that both precision and recall are only slightly better for short methods compared to long methods, while recall is better for long methods. There is little difference in opinion between novices and advanced programmers, suggesting a single automatic system for blank line insertion suffices.

4.8. Threats to Validity

To minimize internal and conclusion threats for studies involving human opinions, we had 3 humans evaluate each method, but clearly having more evaluators per method might change the results. However, it is a compromise between effort and degree to which we minimize the threat.

For internal validity, there could be other factors affecting the comparisons between developer-written and automatically generated blank lines that might show up in the methods we examined. To mitigate this, we randomly chose methods for evaluation from across projects and sizes within our targeted size range. Our evaluation included 39 novices and 15 advanced programmers. The results could vary for other programmers, but our advanced programmers had between 4-20 years of experience.

In terms of external threat, our results may not generalize to other Java programs. To mitigate this, we chose our samples for study from across 7 diverse projects (from 18,186 methods). Our results may not generalize to other languages, but many of the rules are based on basic control structures and data flow common in most procedural and object-oriented languages. We focused on methods between 10-50 lines of code, neither too short for blank line segmentation nor too long to make the human judgement task too tedious. Thus, our results might vary on larger methods.

5. RELATED WORK

Sridhara et al. [18] present a technique to automatically identify groupings of statements (i.e., code fragments) that collectively implement high level actions and synthesize a succinct natural language description to express each high level abstraction. A *high level action* is defined to be a high level abstract algorithmic step of a method. They separately analyze sequences of statements, conditionals, and loops in a method to delineate groupings of statements that collectively implement some high level action, such as *compute max*. This identification does not result in a segmentation of the method body into logically related blocks, but instead finds some blocks that correspond to high level actions.

DOI: 10.1002/smr

There is early work on "beacons" [19, 20], where a beacon can be a well known coding pattern (e.g., 3 lines for swapping array elements), meaningful identifiers, program structure, or comment statements, that signal something to the code reader wants to know about the code segment's functionality. Beacons only represent a name given to a visually recognizable entity or pattern. Similarly, Gil and Maman [21] present a catalog of 27 micro patterns, class-level traceable patterns, similar to design patterns but lower level abstractions. Somewhat related is method extraction, which is primarily based on slicing; block-based slicing [22] or program transformations with slicing [23] to make the dependence-related statements contiguous for extract method refactoring. None of these techniques result in segmenting the method body into logically-related blocks for readability.

Readability metrics help to identify potential areas for improvement to code readability. Through human ratings of readability, Buse et al. [10] developed and automated the measurement of a metric for judging source code snippet readability based on local features that can be extracted automatically from programs. Among of the many features is indentation. More recently, Daryl et al. [6] formulated a simpler lightweight model of software readability based on size and code entropy, which improved upon Buse's approach. Daryl et al. observed indentation correlated to block structure to have a modest beneficial effect according to their results. Other earlier works measured program characteristics with different definitions of readability [24, 25].

Organizations employ coding standards to maintain uniformity across code written by different developers, with the goal of easing program understanding for newcomers to a code [7, 13–16]. Some guidelines [7, 14, 26] have clearly specified the number of blank lines that should be used to separate methods in a class and class fields. They suggest that blank lines should be used to separate different logical sections within methods. However, a logical section is left ill-defined as it is difficult to define precisely.

Automated tools and processes have been constructed with the goal of improving source code readability. These include improving identifier naming [27], generating comments for Java methods [18, 28], refactoring code clones [29], refactoring long methods [30], instituting a readability group to ensure code readability [31], and even adding a development phase in which the program is made more readable [32]. Automatic blank line insertion is complementary to these efforts.

There are other reverse engineering tasks for which formal approaches have been taken. For instance, there has been work in analyzing loops in programs [33], and object equality in Java [34].

6. CONCLUSIONS AND FUTURE WORK

To our knowledge, this is the first automatic system to insert blank lines into source code towards improving code readability and locating points for internal documentation. According to programmers who judged our generated blank lines, the automatically generated blank lines are accurate, and separate different logically-related blocks.

In the future, we will continue to augment our system by examining additional potential code patterns to identify further blocks to be segmented based on programming language design and convention. We will also integrate the techniques into Eclipse and investigate how useful the output is for understanding the method or performing a software maintenance task.

ACKNOWLEDGEMENTS

We thank our human evaluators for all their time, without which the evaluation of SEGMENT would not be possible.

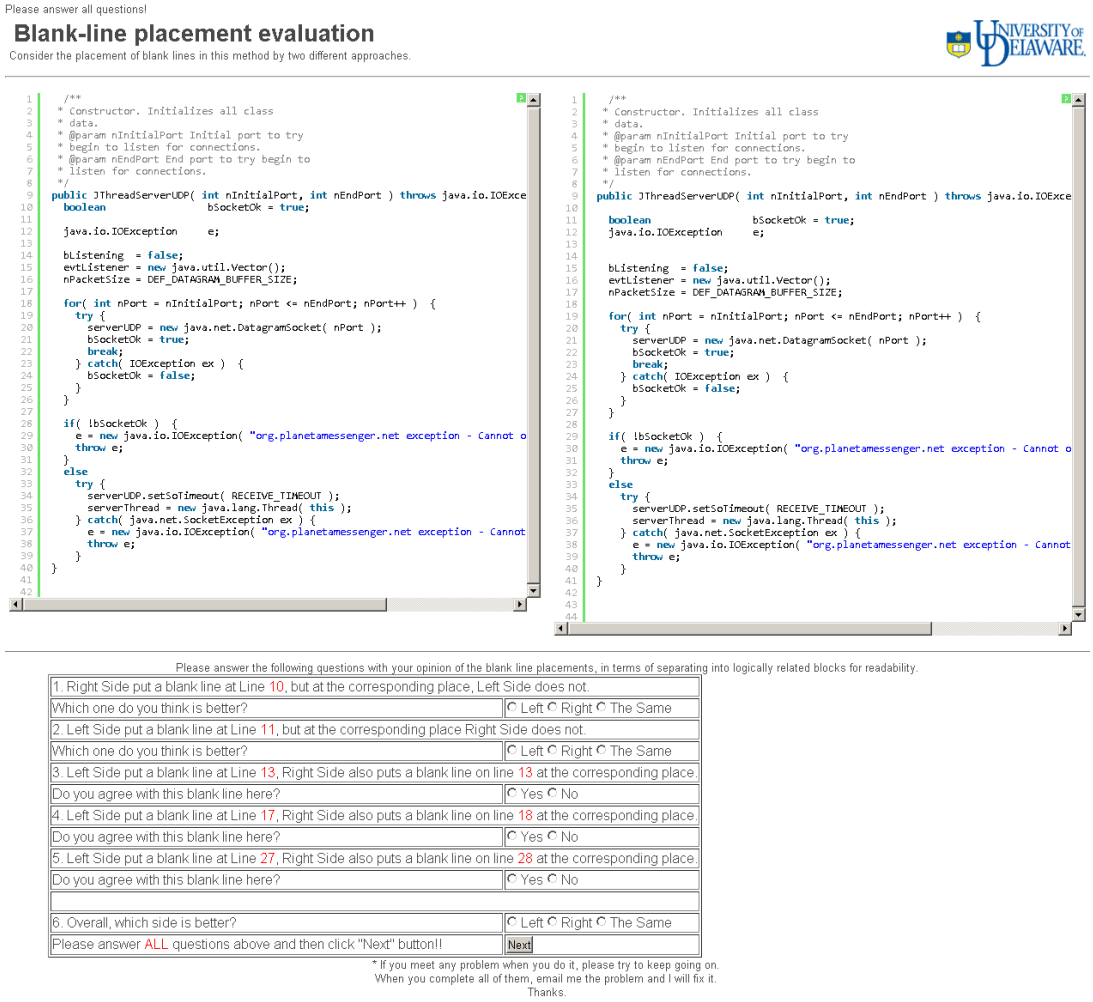
REFERENCES

1. Goldberg A. Programmer as Reader. *IEEE Softw.* 1987; **4**(5):62–70, doi:http://dx.doi.org/10.1109/MS.1987.231775.

2. Murphy G, Kersten M, Robillard M, Cubranic D. The emergent structure of development tasks. *ECOOP 2005 — Object-Oriented Programming, 19th European Conf*, Glasgow, Scotland, 2005.
3. Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, ACM: New York, NY, USA, 2010; 223–226, doi:http://doi.acm.org/10.1145/1810295.1810335. URL http://doi.acm.org/10.1145/1810295.1810335.
4. Buse RP, Weimer WR. Learning a metric for code readability. *IEEE Transactions on Software Engineering* 2010; **36**:546–558, doi:http://doi.ieeecomputersociety.org/10.1109/TSE.2009.70.
5. Jackson S, Devanbu P, Ma KL. Stable, flexible, peephole pretty-printing. *Science of Computer Programming* 2008; **72**(1-2):40 – 51, doi:DOI:10.1016/j.scico.2007.11.002. URL http://www.sciencedirect.com/science/article/pii/S0167642308000440, special Issue on Second issue of experimental software and toolkits (EST).
6. Posnett D, Hindle A, Devanbu P. A simpler model of software readability. *Proceeding of the 8th working conference on Mining software repositories*, MSR '11, ACM: New York, NY, USA, 2011; 73–82, doi:http://doi.acm.org/10.1145/1985441.1985454. URL http://doi.acm.org/10.1145/1985441.1985454.
7. SUN. Code conventions for the java programming language Apr 1999. URL http://www.oracle.com/technetwork/java/codeconvtoc-136057.html.
8. Schach SR. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math, 2004.
9. Bruce K, Danyluk A, Murtagh T. *Java: An Eventful Approach*. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 2005.
10. Buse RP, Weimer WR. A metric for software readability. *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, ACM: New York, NY, USA, 2008; 121–130, doi:http://doi.acm.org/10.1145/1390630.1390647. URL http://doi.acm.org/10.1145/1390630.1390647.
11. Wang X, Pollock L, Vijay-Shanker K. Automatic segmentation of method code into meaningful blocks to improve readability. *Working Conference on Reverse Engineering (WCRE)*, 2011.
12. Jan 2011. URL http://sourceforge.net/.
13. Humphrey W. *Introduction to the personal software process(sm)*. First edn., Addison-Wesley Professional, 1996.
14. Zograf B. Java programming style guidelines Jan 2011. URL http://geosoft.no/development/javastyle.html.
15. Lea D. Draft java coding standard Feb 2000. URL http://gee.cs.oswego.edu/dl/html/javaCodingStd.html.
16. Sutter H, Alexandrescu A. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.
17. Syntaxhighlighter July 2010. URL http://alexgorbatchev.com/SyntaxHighlighter.
18. Sridhara G, Pollock L, Vijay-Shanker K. Automatically detecting and describing high level actions within methods. *Intl Conf on Software Engineering (ICSE'11)*, 2011. To Appear.
19. Brooks R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 1983; **18**:543–554.
20. Crosby ME, Scholtz J, Wiedenbeck S. The roles beacons play in comprehension for novice and expert programmers. *Proceedings of the 14th Annual Psychology of Programming Workshop*, Psychology of Programming Interest Group: London, United Kingdom, 2002.
21. Gil JY, Maman I. Micro patterns in java code. *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM: New York, NY, USA, 2005; 97–116, doi:http://doi.acm.org/10.1145/1094811.1094819.
22. Tsantalis N, Chatzigeorgiou A. Identification of extract method refactoring opportunities. *Software Maintenance and Reengineering, European Conference on* 2009; **0**:119–128, doi:http://doi.ieeecomputersociety.org/10.1109/CSMR.2009.23.
23. Komondoor R, Horwitz S. Effective, automatic procedure extraction. *Program Comprehension, 2003. 11th IEEE International Workshop on*, 2003; 33 – 42, doi:10.1109/WPC.2003.1199187.
24. Aggarwal K, Singh Y, Chhabra J. An integrated measure of software maintainability. *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, 2002; 235 –241, doi:10.1109/RAMS.2002.981648.
25. Borstler J, Caspersen M, Nordstrom M. Toward a Measurement Framework for Example Program Quality. *Department of Computing Science, Umea University*, 2008.
26. Haahr P. A programming style for java Oct 1999. URL http://192.220.96.201/essays/java-style/typography.html.
27. Relf P. Tool assisted identifier naming for improved software readability: an empirical study. *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005; 10 pp., doi:10.1109/ISESE.2005.1541814.
28. Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for java methods. *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, ACM: Antwerp, Belgium, 2010; 43–52, doi:http://doi.acm.org/10.1145/1858996.1859006. URL http://doi.acm.org/10.1145/1858996.1859006.
29. Higo Y, Kusumoto S, Inoue K. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice* 2008; **20**(6):435–461, doi:10.1002/smr.394. URL http://dx.doi.org/10.1002/smr.394.
30. Yang L, Liu H, Niu Z. Identifying fragments to be extracted from long methods. *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, 2009; 43 –49, doi:10.1109/APSEC.2009.20.
31. Haneef NJ. Software documentation and readability: a proposed process improvement. *SIGSOFT Softw. Eng. Notes* May 1998; **23**:75–77, doi:http://doi.acm.org/10.1145/279437.279470. URL http://doi.acm.org/10.1145/279437.279470.
32. Elshoff JL, Marcotty M. Improving computer program readability to aid modification. *Commun. ACM* August 1982; **25**:512–521, doi:http://doi.acm.org/10.1145/358589.358596. URL http://doi.acm.org/10.1145/

358589.358596.
33. Abd-El-Hafiz SK, Basili VR. A knowledge-based approach to the analysis of loops. *IEEE Trans. Softw. Eng.* May 1996; **22**(5):339–360, doi:10.1109/32.502226. URL http://dx.doi.org/10.1109/32.502226.
34. Rupakheti C, Hou D. An abstraction-oriented, path-based approach for analyzing object equality in java. *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 2010; 205 –214, doi:10.1109/WCRE.2010.30.

A. BLANK-LINE PLACEMENT ONLINE SURVEY



Please answer all questions!

Blank-line placement evaluation

Consider the placement of blank lines in this method by two different approaches.

UNIVERSITY OF DELAWARE

```
  /**
   * Constructor. Initializes all class
   * data.
   * @param nInitialPort Initial port to try
   * begin to listen for connections.
   * @param nEndPort End port to try begin to
   * listen for connections.
   */
public JThreadServerUDP( int nInitialPort, int nEndPort ) throws java.io.IOExce
  boolean               bSocketOk = true;

  java.io.IOException     e;

  bListening  = false;
  evtListener = new java.util.Vector();
  nPacketSize = DEF_DATAGRAM_BUFFER_SIZE;

  for( int nPort = nInitialPort; nPort <= nEndPort; nPort++ )  {
    try {
      serverUDP = new java.net.DatagramSocket( nPort );
      bSocketOk = true;
      break;
    } catch( IOException ex )  {
      bSocketOk = false;
    }
  }

  if( !bSocketOk )  {
    e = new java.io.IOException( "org.planetamessenger.net exception - Cannot o
    throw e;
  }
  else
    try {
      serverUDP.setSoTimeout( RECEIVE_TIMEOUT );
      serverThread = new java.lang.Thread( this );
    } catch( java.net.SocketException ex ) {
      e = new java.io.IOException( "org.planetamessenger.net exception - Cannot
      throw e;
    }
}
```

```
  /**
   * Constructor. Initializes all class
   * data.
   * @param nInitialPort Initial port to try
   * begin to listen for connections.
   * @param nEndPort End port to try begin to
   * listen for connections.
   */
public JThreadServerUDP( int nInitialPort, int nEndPort ) throws java.io.IOExce

  boolean               bSocketOk = true;
  java.io.IOException     e;


  bListening  = false;
  evtListener = new java.util.Vector();
  nPacketSize = DEF_DATAGRAM_BUFFER_SIZE;

  for( int nPort = nInitialPort; nPort <= nEndPort; nPort++ )  {
    try {
      serverUDP = new java.net.DatagramSocket( nPort );
      bSocketOk = true;
      break;
    } catch( IOException ex )  {
      bSocketOk = false;
    }
  }

  if( !bSocketOk )  {
    e = new java.io.IOException( "org.planetamessenger.net exception - Cannot o
    throw e;
  }
  else
    try {
      serverUDP.setSoTimeout( RECEIVE_TIMEOUT );
      serverThread = new java.lang.Thread( this );
    } catch( java.net.SocketException ex ) {
      e = new java.io.IOException( "org.planetamessenger.net exception - Cannot
      throw e;
    }
}
```

Please answer the following questions with your opinion of the blank line placements, in terms of separating into logically related blocks for readability.

1. Right Side put a blank line at Line 10, but at the corresponding place, Left Side does not.	
Which one do you think is better?	○ Left ○ Right ○ The Same
2. Left Side put a blank line at Line 11, but at the corresponding place Right Side does not.	
Which one do you think is better?	○ Left ○ Right ○ The Same
3. Left Side put a blank line at Line 13, Right Side also puts a blank line on line 13 at the corresponding place.	
Do you agree with this blank line here?	○ Yes ○ No
4. Left Side put a blank line at Line 17, Right Side also puts a blank line on line 18 at the corresponding place.	
Do you agree with this blank line here?	○ Yes ○ No
5. Left Side put a blank line at Line 27, Right Side also puts a blank line on line 28 at the corresponding place.	
Do you agree with this blank line here?	○ Yes ○ No
6. Overall, which side is better?	○ Left ○ Right ○ The Same
Please answer ALL questions above and then click "Next" button!!	Next

* If you meet any problem when you do it, please try to keep going on.
When you complete all of them, email me the problem and I will fix it.
Thanks.

Current Evaluator: Test

Appendix I: Online survey

DOI: 10.1002/smr