Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability

Xiaoran Wang, Lori Pollock, and K. Vijay-Shanker
Computer and Information Sciences
University of Delaware
Newark, DE 19716 USA
{xiwang, pollock, vijay}@cis.udel.edu

***Abstract*—With the goal of increasing program readability for easier understanding, coding guidelines often include formatting standards such as indenting loop and conditional branch body statements. Similarly, good programming practice suggests that programmers use blank lines to visibly delineate between code segments that represent different algorithmic steps or high level actions. Unfortunately, programmers do not always follow these guidelines. While editors and IDEs can easily indent code based on syntax, they do not currently support automatic blank line insertion, which presents more significant challenges involving the semantics.**

This paper presents 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 an evaluation of the effectiveness of SEGMENT based on developers' opinions. 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.

***Keywords*-Program understanding, readability, software tool, automatic formatting.**

I. Introduction

Software maintenance involves the integrated use of source code and other software artifacts written as English documents [1]. A major time-consuming activity during software maintenance is reading source code [2]–[4] 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. [5]. This observation led to the invention of prettyprinters [6], 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 readability involves the syntactic appearance of code, poor readability is perceived as a barrier to program understanding, which focuses on the semantics [7].

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 [8], blank lines improve readability. Software engineering books [9], [10] suggest using plenty of blank lines to break up big blocks of code. Recent studies with humans judging software readability [11] 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.

The main contributions of this paper are:

- Characterizations of kinds of meaningful blocks composed of consecutive statements,

This material is based upon work supported by the National Science Foundation Grant No. CCF-0702401 and CCF-0915803.

- An algorithm and set of heuristics to automatically segment a large sequence of statements into meaningful blocks, and
- Results from two human judgement studies that indicate strong positive overall opinion of SEGMENT's effectiveness.

II. 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 characterized the common contexts of developers' blank lines, some of which indeed follow Java code conventions [8]. Common uses of vertical spacing in our corpus include:

- After all local declarations at the top of a method body
- 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. 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. Note how the readability of the code is affected if none of the blank lines were present.

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

```
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 }
```

```
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 }
```

Fig. 1. *Example Uses of Blank Lines for Enhanced Readability*

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.

III. Automatic Segmentation

Our approach to automatically inserting blank lines into method code follows the process shown in Figure 2. The

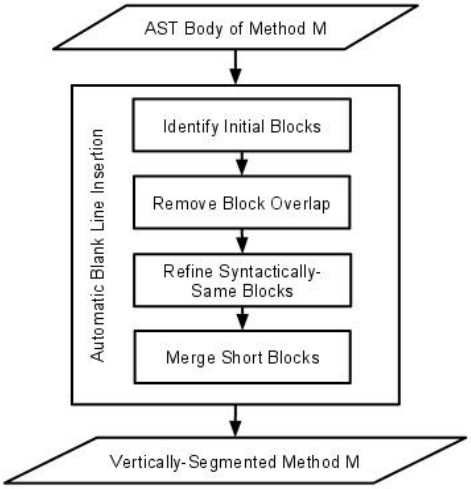

Fig. 2. *Automatic Method Segmentation Process*

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.

```
setShowGrid(true);
setIntercellSpacing(new Dimension(1,1));
tree.setRootVisible(false);
tree.setShowsRootHandles(true);
TreeCellRenderer r = tree.getCellRenderer();
r.setOpenIcon(null);
r.setClosedIcon(null);
r.setLeafIcon(null);
```

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

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

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

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

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.

A. 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 [8].

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 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.

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

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 ...;

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 );
```

- *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;
```

B. 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 Listing 1, 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);
```

Listing 1. 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. For example, 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. 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 in Table I , while RHS expression refers to the right hand side of an assignment or initialization statement.

TABLE II
STATEMENT-PAIR SIMILARITY LEVELS IN A SYNS BLOCK

Syntactic type	**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.

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. There are several cases to consider in computing the similarity of the identifiers. 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.

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.

In general, 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*

---

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 Listing 2, 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.

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

Listing 2. Example (DFC DFC) Consecutive Blocks; Line 4 Overlap

(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 Listing 3, 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);
```

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

Similarly, in Listing 4, 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;
```

Listing 4. 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 Listing 5, 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);
```

Listing 5. Placing Overlapping Statements using Highly Similar Statements

(DFC, E-SWIFT) Pair. 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 Listing 6, 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 }
```

Listing 6. Example (DFC E-SWIFT) Consecutive Blocks; Line 3 Overlap

C. 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 Listing 7, 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);
```

Listing 7. 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 Listing 8. 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();
```

Listing 8. Example of SynS Statement Clustering

D. 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. Listing 9 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.

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

Listing 9. 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

IV. EVALUATION

We implemented the automatic blank line insertion algorithm described in Section III 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?
- insert at appropriate program locations?

We designed two studies to explore these questions. The first study compares 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. However, 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. For this reason, we developed our second study in which blank lines are inserted by people (newcomers to the code) reading the method code. In contrast to the first study, the blocks are segmented from a reader's perspective.

Both studies involve human judges. 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 evaluators 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 developed our studies to have each method to be rendomly assigned to 3 human judges and take the majority opinions when there was no unanimous opinion.

We engaged 15 human evaluators as our subjects, including 6 software engineers, 1 faculty member, 6 graduate students, and 2 undergraduates. Evaluators have programming experience with C++/Java ranging from 4 to 20 years, with a median of 7 years. Nine of the evaluators considered themselves as expert or advanced programmers. Six evaluators have software industry experience ranging from 1 to 10 years. None of the authors participated in the evaluation.

For both studies, we chose two different sets of 50 methods randomly from 7 projects (from 18186 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 40 lines, neither too short for blank line segmentation nor too long to make the task too tedious.

TABLE IV
SUBJECT PROJECTS IN STUDY. KLOC: 1000 LINES OF CODE

Project	#methods	kloc
GanttProject	4956	60
Jajuk	2139	44
PlanetaMessenger	1142	22
Dguitar	1211	20
DrawSWF	2747	41
JRobin	1913	30
SweetHome3D	4078	73

The same evaluators participated in both studies. In the first study, the human evaluators examined the output of SEGMENT and the developers' usage of blank lines. Thus, to reduce the potential bias that the first study may have on human opinions in the second study, evaluators completed the second study before they started on the first study.

A. Developer-written vs. SEGMENT-inserted

Procedure. In the first study, humans were presented with two copies of the 50 methods, one with SEGMENT-inserted blank lines and one with 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 [13]. To reduce possible bias, our web system randomly places the SEGMENT-inserted and developer-inserted copies on the screen, and randomly orders each evaluator's 10 methods to avoid learning effect.

To enable detailed analysis of situations, we manually categorized the differences between the blank line insertion into 4 types:

- Type 1: SEGMENT inserts a blank line; developer does not.
- Type 2: Developer inserts a blank line; SEGMENT does not.
- Type 3: Developer and SEGMENT insert a blank line at different locations, i.e., for the same group of statements, the developer inserted at some location, but SEGMENT inserts at another nearby location. For example, in Listing 10, if SEGMENT inserts between lines 2 and 3, while the developer inserts between lines 3 and 4, then this is called a Type 3 difference.
- Type 4: Developer and SEGMENT insert a blank line at the same location.

```
1 Vector chars = new Vector();
2 byte[] aChar = new byte[1];
3 int num = 0;
4 while( ( num = in.read( aChar )) == 1 )
```

Listing 10. Type 3 Difference

When there was a difference between the system and developers as in types 1-3, 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. When a blank line was inserted by both SEGMENT and developer in the same location (type 4), we asked if they agreed with the blank line.

Results and Discussion. The SEGMENT system inserted blank lines in the same locations as the original developers in 128 of 247 locations analyzed (i.e., 52% of the locations were Type 4). The human evaluators agreed with 127 of the common 128 placements, 99.2% of the common occurrences. In all the 119 cases where SEGMENT differed in some way from the original developer in blank line insertion (Types 1-3), the human evaluators preferred the SEGMENT system in 91 situations, 79.1% of the time. Overall, when we asked the human evaluators to judge the blank line insertion as a whole for entire methods, they reported 33 of 50 instances where SEGMENT was preferred versus 10 of 50 instances where the developer's was preferred overall, and 4 of 50 equally well between the developer and system. Thus, SEGMENT output for entire method segmentation was at least as good or better in 70% of the methods examined.

We believe these results are very encouraging. In 79.1% of the cases where SEGMENT differed from developers, SEGMENT insertions were found to be better than developers. And, overall in 128+91=219 cases out of 247(i.e. 88.7%), the system-generated blank lines are as good as or better than those inserted by developers. Recall these methods all had some blank lines and also do not represent cases where developers were oblivious to vertical spacing considerations.

We analyzed 24 locations where the majority of human evaluation judged developers' insertions to be better than SEGMENT. We found nearly all of these cases fall into two general cases.

Nine of twelve Type 1 cases (where SEGMENT insertion was judged to be extraneous) occur in the preamble of a method. The human judges (as well as the developers) expect to see lines of variable declarations and initializations kept together, and apparently do not believe that it is necessary to insert blank lines in this part (as long as it is not long). However, the insertion of SEGMENT does not distinguish between different general regions of a method. For example, SEGMENT separates the following two statements because there is neither similarity between them nor will they be a SynS block (because the first statement is init and second statement is a declare.) If we don't distinguish between init & declare statements and allow for such blocks in the preamble, nearly all these cases will be handled similar to developers.

```
boolean  bSocketOk = true;

java.io.IOException e;
```

We also examined six of the ten cases for the Type 2 differences (developer has inserted blanks but SEGMENT does not). These cases were at the beginning of the method. SEGMENT does not insert blank lines after a method's signature; more specifically, after '{' and before the method body. However, in these six cases, because of the number of parameters in the signature and the substantial preamble block, the developers inserted blank lines to separate the method signature from the body (see example below) and the evaluators agreed with this vertical spacing.

```
public void onRegisterNewUser( java.lang.String
    strNickName, java.lang.StringBuffer strNewUserId
    , java.lang.String strPasswd )  {

userDetails.setNick( strNickName );
userDetails.setPassword( strPasswd );
session.setUser( userDetails );
```

Among the remaining few cases, the issue was mostly that our current implementation does not cover certain cases, although SEGMENT's underlying approach could have covered these cases.

B. Gold Standard vs. SEGMENT-inserted

Procedure. To obtain a gold standard for blank line insertions, we gave evaluators a set of 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 3 separate judgements for each method. Thus, we obtained 150 independent method annotations totally. To control for any learning effects, each annotator's 10 methods were shuffled and shown in different orders. Also, the 15 evaluators were randomly put into 5 groups.

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 inserted a blank line.

Results and Discussion. The results of comparing SEGMENT's blank line insertion against this gold standard are shown in Table V. The results quite strongly suggest that SEGMENT's automatic insertion agrees with the gold standard, 161/179=89.9% of the gold standard insertions are also locations where SEGMENT inserts blank lines.

TABLE V
AUTOMATIC BLANK LINE INSERTION VERSUS GOLD STANDARD

	Human Majority	**SEGMENT-inserted**	**Percentage**
3/3 Agree	93	92	98.2%
2/3 Agree	86	69	80.2%
Total	179	161	89.9%

SEGMENT inserted 234 blank lines in total. Thus, there are 73 blank lines where the majority of human judges did not

insert any blank line. However, in 44 among those positions, one of the human judges had inserted a blank line where SEGMENT did. So, there are 29 positions where SEGMENT inserted a blank line where none of the human judges did.

We analyzed these 29 cases and again observed two major kinds of situations. 13 of these 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 at all, but evaluators put them together.

```
flags |= 0x2005;

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

The second major type of error cases is due to the fact that SEGMENT makes its decisions to insert blank lines oblivious of location in the code. Six of the 29 locations were inserted blank lines after '}' where it appears there is already sufficient vertical spacing and an additional blank line was not necessary. For example below, consider the blank line before `dispose` statement in the code. As noted before, SEGMENT inserts a blank line after the try block. However, because the `dispose` statement is a single-line block followed by just a '}' in a line by itself and another blank line, none of the human judges inserted a blank line above the `dispose` statement. We believe that Phase IV should consider merging of such single-line blocks.

```
 if (prnJob.printDialog(attr)) {
   try {
      ...
   } catch (Exception PrintException) {
      ...
   }

   dispose();
 }

  setCursor(Cursor.getPredefinedCursor());
}
```

V. Related Work

Sridhara et al. [14] 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.

There is early work on "beacons" [15], [16], 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 [17] 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 [18] or program transformations with slicing [19] 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. [11] 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. [7] 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 [1], [20].

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 [8], [21]–[24]. Some guidelines [8], [22], [25] 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 [26], generating comments for Java methods [14], [27], refactoring code clones [28], refactoring long methods [29], instituting a readability group to ensure code readability [30], and even adding a development phase in which the program is made more readable [31]. Automatic blank line insertion is complementary to these efforts.

VI. 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.

Our first evaluation study shows that 88.7% of the time, the system-generated blank lines are as good as original developer-written ones or even better. The second evaluation study suggests that blank lines generated by the automated system match the majority of human opinions of where vertical spacing should be used.

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.

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.

REFERENCES

[1] K. Aggarwal, Y. Singh, and J. Chhabra, "An integrated measure of software maintainability," in *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, 2002, pp. 235 –241.

[2] A. Goldberg, "Programmer as Reader," *IEEE Softw.*, vol. 4, no. 5, pp. 62–70, 1987.

[3] G. Murphy, M. Kersten, M. Robillard, and D. Cubranic, "The emergent structure of development tasks," in *ECOOP 2005 — Object-Oriented Programming, 19th European Conf*, Glasgow, Scotland, July 27–29, 2005.

[4] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 223–226. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810335

[5] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, pp. 546–558, 2010.

[6] S. Jackson, P. Devanbu, and K.-L. Ma, "Stable, flexible, peephole pretty-printing," *Science of Computer Programming*, vol. 72, no. 1-2, pp. 40 – 51, 2008, special Issue on Second issue of experimental software and toolkits (EST). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642308000440

[7] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," in *Proceeding of the 8th working conference on Mining software repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1985441.1985454

[8] SUN. (1999, Apr.) Code conventions for the java programming language. [Online]. Available: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

[9] S. R. Schach, *Object-Oriented and Classical Software Engineering*. McGraw-Hill Science/Engineering/Math, 2004.

[10] K. Bruce, A. Danyluk, and T. Murtagh, *Java: An Eventful Approach*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2005.

[11] R. P. Buse and W. R. Weimer, "A metric for software readability," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390647

[12] (2011, Jan.). [Online]. Available: http://sourceforge.net/

[13] (2010, July) Syntaxhighlighter. [Online]. Available: http://alexgorbatchev.com/SyntaxHighlighter

[14] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Intl Conf on Software Engineering (ICSE'11)*, 2011, to Appear.

[15] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543–554, 1983.

[16] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "The roles beacons play in comprehension for novice and expert programmers," in *Proceedings of the 14th Annual Psychology of Programming Workshop*. London, United Kingdom: Psychology of Programming Interest Group, Jun. 2002.

[17] J. Y. Gil and I. Maman, "Micro patterns in java code," in *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2005, pp. 97–116.

[18] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," *Software Maintenance and Reengineering, European Conference on*, vol. 0, pp. 119–128, 2009.

[19] R. Komondoor and S. Horwitz, "Effective, automatic procedure extraction," in *Program Comprehension, 2003. 11th IEEE International Workshop on*, may 2003, pp. 33 – 42.

[20] J. Borstler, M. Caspersen, and M. Nordstrom, "Toward a Measurement Framework for Example Program Quality," in *Department of Computing Science, Umea University*, 2008.

[21] W. Humphrey, *Introduction to the personal software process(sm)*, 1st ed. Addison-Wesley Professional, 1996.

[22] B. Zograf. (2011, Jan.) Java programming style guidelines. [Online]. Available: http://geosoft.no/development/javastyle.html

[23] D. Lea. (2000, Feb.) Draft java coding standard. [Online]. Available: http://gee.cs.oswego.edu/dl/html/javaCodingStd.html

[24] H. Sutter and A. Alexandrescu, *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[25] P. Haahr. (1999, Oct.) A programming style for java. [Online]. Available: http://192.220.96.201/essays/java-style/typography.html

[26] P. Relf, "Tool assisted identifier naming for improved software readability: an empirical study," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, nov. 2005, p. 10 pp.

[27] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. Antwerp, Belgium: ACM, 2010, pp. 43–52. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859006

[28] Y. Higo, S. Kusumoto, and K. Inoue, "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*, vol. 20, no. 6, pp. 435–461, 2008. [Online]. Available: http://dx.doi.org/10.1002/smr.394

[29] L. Yang, H. Liu, and Z. Niu, "Identifying fragments to be extracted from long methods," in *Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, dec. 2009, pp. 43 –49.

[30] N. J. Haneef, "Software documentation and readability: a proposed process improvement," *SIGSOFT Softw. Eng. Notes*, vol. 23, pp. 75–77, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279437.279470

[31] J. L. Elshoff and M. Marcotty, "Improving computer program readability to aid modification," *Commun. ACM*, vol. 25, pp. 512–521, August 1982. [Online]. Available: http://doi.acm.org/10.1145/358589.358596