プログラム文並び替えに基づくソースコードの可読性向上の試み

佐々木 唯[1,a)]　肥後 芳樹[1,b)]　楠本 真二[1,c)]

概要：ソフトウェア保守を行うにあたって, 最も時間的コストの高い作業はソースコードを読み理解することである. そのため, ソースコードの可読性を向上させることで, 保守作業全体のコストを削減できる. 本論文では，ソースコードの可読性を向上させるためのプログラム文並び替え手法を提案する．既存研究として, ソースコードを読んで理解しようとする際, 変数が定義されてから参照されるまでの距離が離れていると理解するためのコストが増大することが報告されている. 従って, 文の並びはソースコードの可読性に影響を及ぼすと考えられる. そこで, 変数の定義と参照の間の距離に着目して, モジュール内の文を並び替える手法を提案する. 提案手法をオープンソースソフトウェアに適用し, 並び替えの行われたメソッドについて被験者からの評価を得たところ, 並び替えの行われたメソッドは可読性が向上するという結果が得られた.

キーワード：プログラム理解，ソースコード解析，リファクタリング

An Approach to Improving Readability of Source Code Based on Reordering Program Statements

SASAKI YUI[1,a)]　HIGO YOSHIKI[1,b)]　KUSUMOTO SHINJI[1,c)]

Abstract: Understanding program source code is the most time-consuming task in software maintenance. If readability of source code gets better, we spend less time to understand it. That means we can conduct more efficent software maintenance on source code whose readability is better. In this paper, we propose a technique to reorder program statements without changing its execution behavior for improving readability of source code. Previous research efforts reported that, if a program statement referencing a variable is far from a statement defining the variable, understanding the function that they implement becomes more time-consuming task. Those results indicate that the order of program statements has an impact on readability of source code. Consequently, we focus on distances between definitions and references of variables in source code, and propose a way to reorder them with tactics to shorten the distances. We have conducted an experiment on Java open source software with 44 subjects, and confirmed that most of reordered methods had better readability than their original ones.

Keywords: Program Comprehension, Source code analysis, Refactoring

1. はじめに

近年，ソフトウェアの大規模化，複雑化に伴い，ソフトウェア保守に要する作業量が増大している．ソフトウェアライフサイクルにおいて，保守作業量が占める割合は非常に高い [3]．更に，保守の全行程の中で最も時間的コストの高い作業は，ソースコードを読み理解することであるといわれている [8], [11], [12].

そのため，ソースコードの可読性を向上させることは保守作業全体のコスト削減に繋がると考えられ，プログラム理解に関する研究がこれまでに多く行われている．Buse と Weimer は，テキストの理解のしやすさを**可読性**と定義し

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[a)] s-yui@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

た上で，ソースコードの可読性を測定するため，識別子，特定の記号，インデントや空行などのフォーマット，コメントといったソースコード上の様々な特徴との相関について調査を行った [4]．また，ソースコードの可読性を向上させるために開発者が守るべきコーディング規約をまとめたものとして，Java Code Convensions などが存在する [5]．例えば，メソッドや変数は使用意図が分かるよう，長すぎず短すぎない命名を行うべきであると述べられている．また，適切な空行も可読性を向上させる重要な要因の 1 つであると述べられており，Buse と Weimer の調査結果においても，空行はコメントよりも重要度が高いことが分かっている．また，Wang らは，ソースコード中の文から意味のあるまとまりを識別し，その間に空行を挿入することでソースコードの可読性を向上させる手法を提案している [17]．

このように，ソースコードの可読性を向上させるための手法やツールは多く存在し，これらはリファクタリング操作として考えることができる．リファクタリングとは，プログラムの振る舞いを変えずに内部構造を変化させる技術である．ソースコードには将来的に問題を引き起こす可能性のある「不吉なにおい」が存在し，保守性を低下させる原因であるといわれている．そのような不吉なにおいに対する具体的な対処方法は，Fowler によってリファクタリングパターンとしてまとめられている [7]．

本研究では，ソースコードの可読性を向上させるためのリファクタリング手法を新たに提案する．Buse と Weimer の調査結果では，ソースコード可読性に最も影響を与えている要因は識別子の数であった [4]．また，変数が定義されてから参照されるまでの間に多くの処理を含む場合，理解するためのコストが増大するという報告もある [13]．このような状況を改善するためには，変数のスコープを狭める，変数の代入文を参照される直前まで移動するといった文の並び替え操作が有効である．ソースコードの可読性を向上させるためのリファクタリング手法として，空行の挿入やインデントの整形などが行われている [9], [17]．

本研究では，変数の定義と参照の間の距離に着目して，モジュール内の文を並び替える手法を提案する．提案手法をオープンソースソフトウェアに適用したところ，提案手法によって並び替えの行われたモジュールは可読性が向上したという結果が得られた．

本研究の主な貢献を以下に記す．

- プログラムの振る舞いを変えることなく，ソースコードの可読性を向上させる手法を提案した．提案手法では，変数を内側のブロックに移動させてスコープを小さくする，およびブロック内での文の入れ替えにより定義と参照の間の距離を短くする，の 2 つの戦略に基づいてソースコード中の文の順番を入れ替える．
- 44 名の被験者に協力して頂き，提案手法の評価を行った．その結果，実験対象とした 20 のメソッドのうち，16 のメソッドにおいて，提案した並び替え手法によりメソッドの可読性が向上したとの結果を得た．

2. 関連研究

2.1 ソースコードの可読性に関する調査

Buse と Weimer はさまざまなメトリクスとソースコード可読性との相関を調査した結果，識別子の数はソースコード可読性に最も影響を与えていることを示している [4]．また，人はソースコードを理解しようとする際，ソースコードを読みながら頭の中で実行することがある．このような作業をメンタルシミュレーションと呼ぶ [6]．Nakamura らは，人の記憶形式をキューでモデル化し，メンタルシミュレーションのコストを計測した [13]．この結果，キューにない変数を参照するとき，すなわち定義されてから参照されるまでの間に多くの処理を含む変数を参照するとき，コストの増大に繋がるということが分かっている．

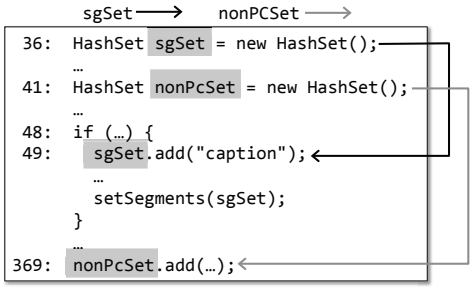
Biegel らは，Java ソースコード中のフィールド及びメソッドの並び順は可読性に影響を与える要因であると考え，その並び順にどのような基準があるか，16 のオープンソースソフトウェアを対象に調査を行った [2]．その結果，最も広く用いられている基準は Java Code Convensions で定められている基準であるが，それに続く基準はさまざまなものが存在することが分かっている．

また，ソースコードを理解するための時間のうち，1 つのドキュメントに対してスクロール操作などで移動を行う時間は約 7 分の 1 を占めるという報告がある [11]．

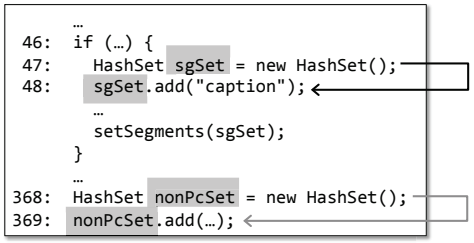
2.2 ソースコードの可読性の向上を目的としたリファクタリング手法

エディタ上での強調表現やフォーマットの整形など，ソースコード上の見た目を改善する技術を Pretty-Printing と呼ぶ [9]．Pretty-Printing は古くから用いられている技術で，プログラミング言語に依存しない手法は Oppen によって最初に提案された [14]．また，Pretty-Printing は基本的にプログラムの構文的な情報を元に行われているが，Wang らは構文情報の他にデータ依存も考慮した上でソースコードの意味的なまとまりを識別し，空行で分割することで可読性を向上させる手法を提案した [17]．

Atkinson と King は，行数や文の数などのメトリクスを用いてリファクタリング候補を自動検出する手法を提案した [1]．Relf は，ソースコードの可読性を高めるために，ソースコード上の情報から適切な識別子名を特定し，提示する手法を提案した [15]．Tsantalis と Chatzigeorgiou は，プログラム中に存在する全ての変数について，データの依存関係を元に関連性のある文のまとまりを特定し，メソッド抽出リファクタリングの候補を提示する手法を提案した [16]．



(a) 移動前



(b) 移動後

図 1 プログラム文の移動例

Fig. 1 An example of reordering program statements

図 2 手順 2 の適用例

Fig. 2 An example of STEP2 application

3. 研究動機

図 1(a) のソースコードは，変数の定義と参照が離れている例を示している．例えば，図 1(a) において，変数 *sgSet* は 48 行目以降の if 文内でのみ参照されているにもかかわらず，外側のブロックで定義されている．一般的に，このように局所的に用いられる変数はスコープを狭めることが望ましい．変数 *nonPcSet* については，定義と参照が同一スコープで行われているためスコープを狭めることはできないが，定義を行う文を参照を行う文の直前に移動することは可能である．これら 2 つの変数について，それらの定義を行う文を移動すると，図 1(b) のようになる．この移動を行うことによって，2 つの変数の定義と参照の間の距離が縮まり，ソースコードの可読性が増すと著者らは考えた．

本研究では，ソースコードの可読性を向上させるために，文の並び替えを行う手法を提案する．図 1 の例に基づいて，以下の 2 つの移動戦略を用いる．

戦略 A 変数のスコープを狭めるため，内部ブロックへ文を移動する．

戦略 B 変数の定義と参照の間の距離を短くするため，共通のブロック内で文を移動する．

4. 提案手法

4.1 変数の定義と参照間の距離の取得

本手法では，変数の定義と参照の関係を **Def-Use チェイン**（以下，**DU チェイン**）から取得する [10]．DU チェインとは，ある変数の定義と参照の関係を表したものである．2 つの文 s_1, s_2 が次の条件を全て満たすとき，文 s_1 から文 s_2 へ DU チェインが存在する．

- 文 s_1 は変数 v を定義する．
- 文 s_2 は変数 v を参照する．
- 文 s_1 から文 s_2 の間には，変数 v の再定義のない 1 つ以上の実行パスが存在する．

変数の定義と参照間の距離は，DU チェインを用いて算出する．ある DU チェイン c の距離 $distance(c)$ を，DU チェイン c によって結ばれた 2 つの文の間に存在する文の数とする．このとき，あるブロック b に含まれる全ての DU チェインの総距離 $totaldistance(b)$ は，b に含まれる全ての DU チェインの集合 $DUchain(b)$ を用いて以下の式で表す．なお，“DU チェイン c がブロック b に含まれる” とは，c を構成する 2 つの文が共に b 内に存在することを表す．

$$totaldistance(b) = \sum_{c \in DUchain(b)} distance(c) \qquad (1)$$

4.2 手法の概要

本手法は，ソースコードを入力として以下の手順で文の並び替えを行う．

手順 1 ソースコードから抽象構文木（AST）を構築する．

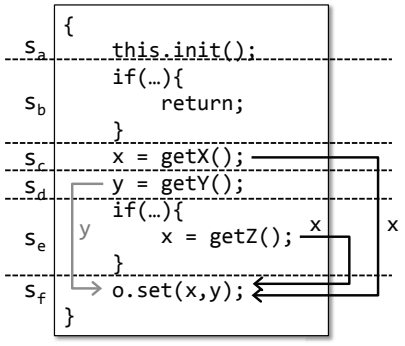
手順 2 AST を後順走査し，訪れたブロックに対して次の 2 つの移動戦略を適用する．

戦略 A 内部ブロックへ文を移動する．

戦略 B 共通のブロック内で文を移動する．

手順 3 走査を終えた AST からソースコードを生成する．

手順 2 において，2 つの移動戦略が適用される様子を図 2 に示す．2 行目の if ブロックに訪れたとき，if ブロック内に移動することで変数のスコープを狭めることのできる文があれば，その文を if ブロック内へ移動する．続いて，if ブロックの *totaldistance* が最小になるよう，ブロック内の文を並び替える．



(a) 並び替え対象の文

制約の表記

a→b: 文bは文aの後に出現しなければならない

Def-Use	Def-Def	Escape
$s_c \rightarrow s_f$ $s_d \rightarrow s_f$ $s_e \rightarrow s_f$	$s_c \rightarrow s_f$	$s_a \rightarrow s_b$ $s_b \rightarrow s_c$ $s_b \rightarrow s_d$ $s_b \rightarrow s_e$ $s_b \rightarrow s_f$

(b) 図 3(a) に対する順序制約

図 3 文の順序制約の例

Fig. 3 An example of reordering constraints

4.3 戦略 A の実現方法

以下の全ての条件を満たすとき，文 s はブロック b 内へ移動可能であるとする．

- 文 s は変数宣言文である．
- 文 s はブロック b の外側に存在し，ブロック b 内の文に対して DU チェインが存在する．
- 文 s はブロック b が実行される前に必ず実行される文である．
- 文 s で定義された全ての変数は，ブロック b 内の文でのみ参照される．
- 文 s で定義された全ての変数は，ブロック b が実行される前に再定義されることはない．

上記の条件を全て満たす文が存在すれば，この文をブロック内の先頭要素として配置する．この戦略は文のスコープを狭めることのみを目的としているため，ここでは文をどの位置に配置すべきかということは考慮しない．

4.4 戦略 B の実現方法

着目中のブロック内の文を並び替える際，提案手法では，“プログラムの振る舞いを変えない” という制約を満たす全ての文の並び（順列）を生成し，その中から *totaldistance* が最小のものを選択するという方法を用いる．

例として，図 3(a) のブロックに対して戦略 B を適用する場合を考える．このとき，並び替え対象となる文は，文 S_A から S_F である．ただし，文 S_B や S_E のように，並び替え対象の文自身がブロックである場合も存在する．これらの文から生成される，プログラムの振る舞いを変えないための順序制約は図 3(b) の通りとなる．

プログラムの振る舞いを変えない範囲で文の並び替えを行うために，提案手法では，プログラム文が満たさなければならない順序制約として下記の 3 つを定める．

Def-Use 制約 ある変数の定義と参照の関係にある 2 つの文は，現在の出現順序を保たなければならない．例えば図 3(a) の文 S_F は，文 S_D で定義された変数 y を参照しており，この 2 つの文を入れ替えると参照することができなくなってしまう．

Def-Def 制約 同一変数を定義する文が複数存在するとき，その出現順を保たなければならない．例えば，図 3(a) の文 S_C, S_E ではともに変数 x が定義されており，文 S_F は，このどちらか一方から値を参照することになる．この 2 つの文を入れ替えると，文 S_C が文 S_E による変数 x の定義を上書きしてしまうため，文 S_F は文 S_E で定義された値を参照することができなくなってしまう．

Escape 制約 ブロック外へのジャンプ命令を含む文をまたいで文の移動をすることはできない．例えば，図 3(a) の文 S_B は return 文を含んでいる．このとき，文 S_B より前の文は必ず実行されるが，後の文は文 S_B の条件によっては実行されるとは限らない．よって，文 S_A を文 S_B よりも後に移動することはできないし，文 S_C, S_D, S_E, S_F を文 S_B の前に移動することもできない．なお，ブロック外へのジャンプ命令とは，return 文の他に continue 文, break 文や，Java 言語の場合 throw 文, assert 文が含まれる．

上記の制約を踏まえた上で，ブロック b 内の文に対する並べ替えは下記の手順で行われる．

手順 1 順序制約を満たす全ての順列（文の並び）を生成する．

手順 2 手順 1 で生成した順列の中から $totaldistance(b)$ が最小であるものを抽出する．

しかし，$totaldistance(b)$ が最小である順列が複数存在する場合がある．その場合はさらに下記の処理を行う．

手順 3 抽出した順列の中で，オリジナルの順列（入力メソッドにおける文の並び）と最も近いものを 1 つだけ抽出する．

本研究では，変数の定義と参照の間の距離にのみ着目した文の並び替えを行うため，それ以外の要素で文の順序が入れ替わることは適切でないと考え，このような処理を行う．この処理では，オリジナルの順列と最も近いものを選ぶために，Spearman の順位相関係数を利用する．手順 2 で得られた全ての候補とオリジナルの順列との間で Spearman の順位相関係数を計測し，最も値が高いものを出力とする．

5. 評価実験

提案手法を実装し，評価実験を行った．本実験の目的は，提案手法を用いた文の並び替えによって，ソースコードの

可読性が向上するかを評価することである．実験対象として，Java で記述されたオープンソースソフトウェアである TVBrowser を用いた．

5.1 準備

TVBrowser に含まれる全メソッドに対して，提案手法を適用した．TVBrowser には約 3700 のメソッドが含まれており，そのうちの 215 のメソッドで文の並び替えが行われた．提案手法の評価を行うため，20 のメソッドを抽出し，オリジナルのソースコードと，提案手法を適用した後のソースコードとで，どちらが理解しやすいか被験者に答えてもらうアンケートを実施した．アンケートの準備は下記のように行った．

手順 1 20 の対象メソッドのオリジナルソースコードと並び替え後のソースコードから，コメントおよび空白を取り除く．また，インデントや改行位置などのフォーマットを統一する．

手順 2 各メソッドにつき，オリジナルと並び替え結果をランダムに A, B と区別し，読みやすさについて以下の項目から選んでもらう．

- A の方が読みやすい．
- B の方が読みやすい．
- 読みやすさに違いはない．

上記のアンケートを web 上で公開し[*1]，被験者を募った．その結果，44 名の被験者が実験に参加した．アンケートでは被験者の Java 使用経験について質問を行っており，その結果を表 1 および表 2 に示す．

5.2 結果

実験結果を図 4 に示す．グラフの縦棒は各メソッドを表し，44 名の被験者が選んだ回答の内訳を表示している．また，各内訳の合計値をグラフの右側に記載している．グラフより，「違いがない」という回答を除けば，20 個中 16 個のメソッドで提案手法適用結果のメソッドを「読みやすい」と判断した被験者数が多いという結果が得られた．また，Wilcoxon の符号順位和検定より，提案手法を選んだ人数とオリジナルを選んだ人数には，優位水準 1%で差がある（p 値は 0.002 であった）ことを確認した．以上の結果から，提案手法によってメソッドの可読性を向上させる文の並び替えを行うことができたといえる．

表 1 Java を用いたプログラミングの経験
Table 1 Programming experiments with Java language

内訳	人数
まったく使ったことがない	1 名
1,000 行未満	7 名
1,000~10,000 行程度	23 名
10,000 行以上	13 名

[*1] http://t.co/3gDf5N7

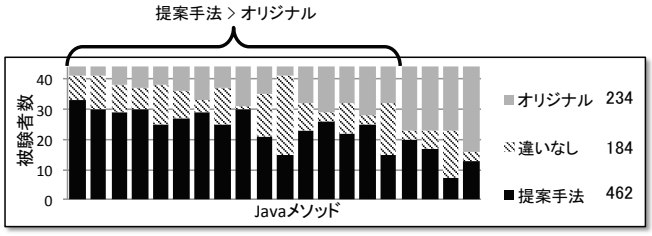
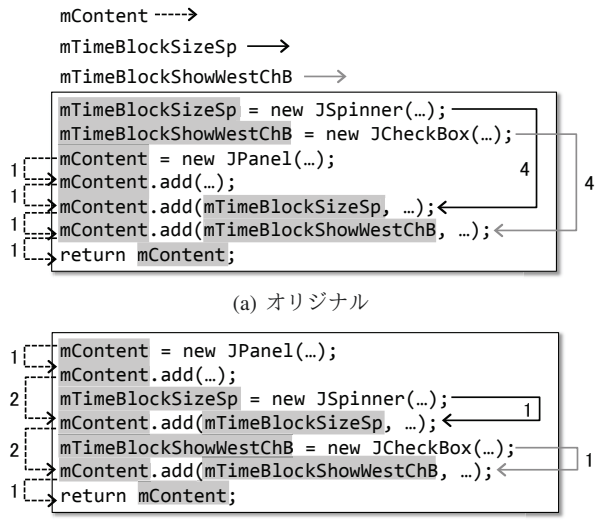


図 4 各対象メソッドに対する回答の内訳
Fig. 4 Questionnaire results for target methods



(a) オリジナル

(b) 提案手法適用後

図 5 オリジナルの方が読みやすいと判断されたメソッド
Fig. 5 A method that its original had a better readability than the reordered one

5.3 考察

図 4 より，4 つのメソッドについてはオリジナルのメソッドを「読みやすい」と判断した被験者が多いことが分かった．この 4 つのメソッドについて調査したところ，以下の特徴が見られた．

- 類似した変数名の宣言が連続して行われている．
- 同一のオブジェクトに対して，同名のメソッド呼び出しが連続して行われている．

例えば，図 5 はオリジナルの方が読みやすいと判断した人数の最も多かったメソッドである．図中の矢印は DU チェインを表し，そのラベルは DU チェインの距離を表し

表 2 Java の使用機会（複数回答あり）
Table 2 Opportunities using Java language

内訳	人数
授業（学生時代）	31 名
研究（学生時代）	28 名
趣味	18 名
仕事	12 名

ている．提案手法の適用によって，このメソッドの先頭で定義されていた 2 つの文が，参照される直前へ移動するという文の並び替えが行われた．しかし，被験者の多くはオリジナルの方が読みやすいと評価している．オリジナルのメソッドには，メソッドの先頭で定義されている 2 つの変数名が類似している，および，変数 *mContent* が指すオブジェクトに対して連続してメソッド呼び出しが行われている，という特徴がある．一方，提案手法適用後のメソッドでは，このような類似する文の並びが保たれていない．

提案手法による文の並び替えを行うことで，これらの類似した文の構造が保たれていない例がいくつか見られた．このことから，類似した文の並びはソースコードの可読性に貢献するものである可能性がある．

6. 妥当性について留意すべき点

5 節の実験では，215 のメソッドに対して文の並び替えが行われた．しかし，この 215 のメソッド全てが，オリジナルのメソッドとの間において振る舞いを保持していたわけではない．TVBrowser はそのソースコード中において，JDK 等の外部ライブラリに含まれるメソッドを呼び出している．そのようなメソッド呼び出しにおいて，オブジェクトの状態が変化しているかは不明である．今回の実験では，外部のメソッド呼び出しがあった場合には，そのメソッドではオブジェクトの状態が変化しないという前提をおいた上で文の並び替えを行った．

しかし，この前提は必ずしも成り立つわけではない．よって，提案手法により並び替えた 215 のメソッドからランダムに 1 つを抽出し，並び替えによって振る舞いが変化していないかどうかを著者らが手作業により確認した．振る舞いが変わっていないと判断した場合には，そのメソッドを実験対象に加えた．このようにして，ランダムに 1 つ抽出し，そのソースコードを調査するという作業を，実験対象が 20 になるまで繰り返した．

7. おわりに

本研究では，変数の定義と参照の間の距離に着目して，ソースコード中の文を並び替える手法を提案した．オープンソースソフトウェアに対して手法を適用し，並び替えが行われた 20 のメソッドを対象に，44 名の被験者が比較を行った．その結果，提案手法を適用することで 16 のメソッドの可読性が向上したことが確認できた．更に，可読性に影響を与える文の並び方には，変数の定義と参照の間の距離だけではなく，類似した文の並びも影響を与える可能性があるという結果が得られた．

謝辞

本研究は，日本学術振興会科学研究費補助金基盤研究 (S)(課題番号：25220003)，挑戦的萌芽研究 (課題番号：24650011)，および文部科学省科学研究費補助金若手研究 (A)(課題番号：24680002) の助成を得て行われた．

参考文献

[1] Atkinson, D. C. and King, T.: Lightweight Detection of Program Refactorings, *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pp. 663–670 (2005).

[2] Biegel, B., Beck, F., Hornig, W. and Diehl, S.: The Order of Things: How Developers Sort Fields and Methods, *Proceedings of the 28th International Conference on Software Maintenance*, pp. 88–97 (2012).

[3] Boehm, B. and Basili, V. R.: Software Defect Reduction Top 10 List, *Computer*, Vol. 34, No. 1, pp. 135–137 (2001).

[4] Buse, R. P. L. and Weimer, W. R.: Learning a Metric for Code Readability, *IEEE Trans. Softw. Eng.*, Vol. 36, No. 4, pp. 546–558 (2010).

[5] Code Conventions for the Java Programming Language: `http://www.oracle.com/technetwork/java/codeconv-138413.html`.

[6] Dunsmore, A. and Roper, M.: A Comparative Evaluation of Program Comprehension Measures (2000).

[7] Fowler, M.: *Refactoring: improving the design of existing code*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999).

[8] Goldberg, A.: Programmer as Reader, *IEEE Softw.*, Vol. 4, No. 5, pp. 62–70 (1987).

[9] Jackson, S., Devanbu, P. and Ma, K.-L.: Stable, flexible, peephole pretty-printing, *Sci. Comput. Program.*, Vol. 72, No. 1-2, pp. 40–51 (2008).

[10] Khedker, U., Sanyal, A. and Karkare, B.: *Data Flow Analysis: Theory and Practice*, CRC Press, Inc., Boca Raton, FL, USA, 1st edition (2009).

[11] Ko, A. J., Myers, B. A., Coblenz, M. J. and Aung, H. H.: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks, *IEEE Trans. Softw. Eng.*, Vol. 32, No. 12, pp. 971–987 (2006).

[12] Murphy, G. C., Kersten, M., Robillard, M. P. and Čubranić, D.: The emergent structure of development tasks, *Proceedings of the 19th European conference on Object-Oriented Programming*, pp. 33–48 (2005).

[13] Nakamura, M., Monden, A., Itoh, T., Matsumoto, K.-i., Kanzaki, Y. and Satoh, H.: Queue-Based Cost Evaluation of Mental Simulation Process in Program Comprehension, *Proceedings of the 9th International Symposium on Software Metrics*, pp. 351–360 (2003).

[14] Oppen, D. C.: Prettyprinting, *ACM Trans. Program. Lang. Syst.*, Vol. 2, No. 4, pp. 465–483 (1980).

[15] Relf, P. A.: Tool assisted identifier naming for improved software readability: an empirical study, *Int'l Symp. on Empirical Software Engineering*, pp. 53–62 (2005).

[16] Tsantalis, N. and Chatzigeorgiou, A.: Identification of Extract Method Refactoring Opportunities, *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pp. 119–128 (2009).

[17] Wang, X., Pollock, L. and Vijay-Shanker, K.: Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability, *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, pp. 35–44 (2011).