Vol.2013-SE-181 No.1
2013/7/17

可読性の評価と改善過程を支援するソースコードビュアー

高橋圭一[†1]

ソースコードの可読性のメトリクスの一つに Buse らのメトリクスがある．Buse らのメトリクスは人間の評価を基準としているため，開発者の感覚に近いスコアが得られる．本稿ではいくつかのソースコードに対して Buse らのメトリクスを適用し，可読性を向上するリファクタリングを支援するための方法をトライアンドエラーで検証した．そこで得られた知見より，開発者がソースコードの可読性を客観的に評価し，可読性の低い部位を効率的に特定する修正作業を支援するソースコードビュアーを提案する．

Source Code Viewer for Evaluating and Improving Code Readability

KEIICHI TAKAHASHI[†1]

Buse *et al.* proposed one of code readability metrics. As the metric is based on human judgments on how easy a text is to understand, the score of the metric can be close to software developers. In this paper we apply the Buse *et al.* metric for some source codes in try-and-error, which is to find how to use the metric to improve the code readability in software refactoring. Consequently, we propose a source code viewer for improving its readability, which enable software developers to identify and improve the lower readability code portions efficiently.

1. はじめに

ソフトウェア開発におけるさまざまなフェーズでソースコードを読んで理解する作業が必要となる．特にソフトウェアライフサイクルの 70%を占めるといわれる保守作業のうち，最も時間を要する作業はソースコードを読む作業であるという研究がある[1][2]. 可読性の高いソースコードを開発することはソフトウェア開発の生産性を高めるばかりでなく品質の向上にも寄与すると考える．可読性を含むソースコードの改善活動にリファクタリングがあるが，納期やコストの制約が厳しいプロジェクトでは，本質的な機能の開発以外に割り当てられる時間は限られている[3]. 近年，Buse らによって提案された，ソースコードの可読性を自動的に判定するメトリクスがある[4]. ソースコードの複雑さを測定するメトリクスには Hastead や McCabe らをはじめとして古くから研究されているが[5][6]，Buse らのメトリクスは人間の評価を取り入れているため，ソフトウェア開発者の直感にしたがった評価ができると期待されている．我々はこの Buse らのメトリクスに着目し，ソフトウェア開発作業の中での活用方法について検討する．本稿では研究の初期段階として，いくつかのソースコードに対して Buse らのメトリクスを適用し，可読性を向上するリファクタリングを支援するための方法を探索的に検証した．その結果から，開発者が開発中のソースコードの可読性を客観的に評価し，可読性の低い部位を効率的に特定し，修正する作業を支援するためのソースコードビュアーを試作した．

本稿の構成は次の通りである．2 章で Buse らのメトリクスの概要について説明し，3 章にて Buse らのメトリクスをオープンソースソフトウェアの 1 つである Junit に適用し，可読性改善のためのリファクタリングに適したスコアの表示方法について検討した予備実験を示す．4 章にて，予備実験から得られた知見を用いたソースコードビュアーを提案する．5 章で評価実験を示し，6 章にて考察を述べ 7 章にてまとめと今後の課題を述べる．

†1 近畿大学
Kinki University

2. Buse らのメトリクス

2.1 特徴

Buse らのソースコードの可読性のメトリクスは，バージニア大学の学生 120 人（1 年生～大学院生）に Java で書かれた 100 個のコード片を評価させたデータを基本としていることが特徴である．この実験に使用したコード片は平均 7.7 行と比較的短い．この理由として Buse らはコード片と人の評価をより強く対応づけるためと主張している．被験者には事前に評価の基準は与えず，個々人の直感的によって評価させた．評価は 5 段階評価で，最も読みにくいコードを 1 とし，最も読みやすいコードを 5 とした．

2.2 人の評価による可読性のスコアの算出方法

個々のコード片に対して，表 1 のソースコード特性をそれぞれ測定する．またコード片に対する人の評価（1～5）の平均値をもとに閾値aとの大小関係で「読みやすい」「読みにくい」の 2 値のいずれかが自動的に決まる．これらソースコードの特性を訓練データとして，また「読みやすい」「読みにくい」の 2 値のいずれかを教師データとして機械学習する．訓練終了後，新たに与えられたコード片について表 1 のソースコード特性を測定し学習モデルに入力し，

a Buse らの実験により，人の評価の平均値の確率密度分布が 3.136 を境界として二峰性をもつことから，人の評価の平均値でコード片を「読みやすい」「読みにくい」の 2 値で評価ができると主張している．

「読みやすい」に分類できる確率を可読性のスコアとして出力する．すなわち，Buse らのメトリクスのスコアは 0 から 1 の値をもつ．

なお, この学習モデルは表 1 のうち#blank lines の平均値, #comments の平均値，#arithmetic operators の平均値が可読性のスコアに正に寄与し，その他の特性は負に寄与することが Buse らによって示されている[4]．したがって，測定対象のソースコード中に空行，コメント，算術演算子が多く含まれていると，そのソースコードの可読性のスコアは高くなる．

表 1 Buse らのメトリクスが使用するソースコード特性
Table 1 Feature names to calculate a metric score of Buse *et al*.

Ave.	Max.	Feature Name
✔	✔	line length (# characters)
✔	✔	# identifiers
✔	✔	identifier length
✔	✔	identation (preceding whitespace)
✔	✔	# keywords
✔	✔	# numbers
✔		# comments
✔		# periods
✔		# commas
✔		# spaces
✔		# parenthesis
✔		# arithmetic operators
✔		# comparison operators
✔		# assignments (=)
✔		# branches (if)
✔		# loops (for, while)
✔		# blank lines
	✔	# ocurrences of any single character
	✔	# occurences of any single identifier

3. 予備実験

Buse らのメトリクスの有効性は Buse らの実験によって示されているが，オープンソースのプロダクト全体に対してメトリクスを適用した結果が示されているだけで，具体的に特定のコード片に適用してメトリクスのスコアがどのように変動するかは示されていない．そこで，既に開発を終えたコードが手元にあるとし，そのコードに対して Buse らのメトリクスを適用し，得られるスコアと可読性向上のための修正方法を探索的に検証する．

3.1 クラス単位の可読性測定

(1) 方法

まず Buse らのメトリクスの感度を調べるため, 具体的なソースコードに対して可読性の計測をおこなう．計測対象は Java の代表的なオープンソフトウェアの 1 つである Junit とする[b]．使用したバージョン 4.12 で，パッケージ junit.framework.*の 15 個のクラスファイルを対象とする[7]．

b Junit は Buse らのモデルの構築時に用いられている. これは予備実験が学習モデルの性能を評価するためではなくコードの修正内容がスコアにできるだけ正しく反映されることを期待したためである．

Buse らのメトリクスは Buse らのホームページで公開されている実行可能な Jar ファイルである readability.jar を用いる[8].

(2) 結果

測定結果を表 2 に示す．スコアが 0.9 を越えるクラスが 5 個あった（表 2 着色部）．これらは可読性が高くリファクタリングの必要がないコードと考えられる．一方，残り 10 個のクラスのスコアはほぼ 0 であり，リファクタリングが必要と考える．

表 2 junit.framework.*の可読性スコア
Table 2 Readability scores of the classes in junit.framework package.

クラス名	スコア
Assert	0
AssertionFailedError	0.92
ComparisonCompactor	0
ComparisonFailure	0.085
JUnit4TestAdapter	0
JUnit4TestAdapterCache	0
JUnit4TestCaseFacade	0.16
package-info	0.999
Protectable	0.999
Test	0.997
TestCase	0
TestFailure	0.009
TestListener	0.977
TestResult	0
TestSuite	0

3.2 ソースコードの分割方法

3.1 の結果，可読性のスコアが 0 付近であることがわかった 10 個のクラスを対象としてリファクタリングをさらに進める．3.1 に示したように，クラス単位のスコアはほぼ 0 であるため，可読性を低下させているコード部位を特定することが困難であり修正作業が非効率的である．そこでクラス中のソースコードを分割して，それぞれの可読性のスコアを計算する方法を考える．

(1) 方法

分割対象であるクラスを構文解析し，メソッドやフィールドを抽出した結果に対して可読性のスコアを計算する方法が考えられるが，インデントやコメントなど機能とは直接関係がない要素も含めて対象としたいので，本稿ではソースコード中の空行を区切り記号としてソースコードを分割する．開発者自身が意図して空行を挿入して分割したコード片のまとまりには意味があると考えるからである．この分割方法の有効性を確かめるために，パッケージ junit.framework.*の 15 個のクラスを空行で分割する．

(2) 結果

空行でのソースコードの分割は構文解析に比べるとシンプルな基準であるが，パッケージ junit.framework.*の 15 個のクラスにおけるすべてのメソッドの抽出に成功した．ま

た，それぞれのメソッドはコメントを含めた状態でソースコードを分割できることがわかった．分割したソースコードの例を図 1 に示す．また，TestSuits.java の addTestsFromTestCase メソッドは 25 行であるが，開発者がメソッド中に空行を 2 カ所挿入しているため比較的短い 3 つコード片に分割できた．さらに，TestCase.java の 7 行目から空行が存在せず 70 行連続したコード片となった．これが 15 個のクラスで最大のコード片であった．ただし，このコード片はほとんどがコメント文であるため分割しなくても問題がないケースと考えられ，開発者自身もそうした意図で空行を挿入しなかったと考える．

```
/**
 * Asserts that a condition is true. If it isn't it throws
 * an AssertionFailedError with the given message.
 */
static public void assertTrue(String message, boolean condition) {
    if (!condition) {
        fail(message);
    }
}
```

図 1　空行によって分割したソースコード例

Figure 1　A example of source code divided by blank lines.

4. 提案手法

3 章の探索的な予備実験をもとにソースコードビュアーのプロトタイプを開発した．以降，ソースコードビュアーの概要と特徴的な機能について説明する．

4.1 概要

ソースコードビュアーのページ遷移図を図 2 に示す．(a)にてローカルディレクトリのパスを入力して，そのパス直下に含まれる Java ソースコードを読み込む．今回はプロトタイプのためサブディレクトリ以下の再帰読み込みはおこなわない．ここで Buse らのメトリクスのスコアを計算する．(b)にてクラス単位の可読性のスコアを一覧表示する．ここでクラスを 1 つ選択することによって，(c)にて空行で分割したソースコードとその可読性のスコアを表示する．図 3 および図 4 に各ページの表示例を示す．図 3 にて AssertionFailedError.java をクリックすると図 4 が表示される．

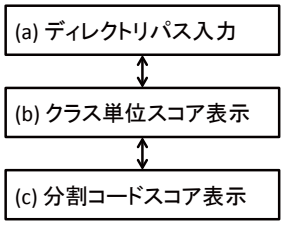

図 2　ソースコードビュアーのページ遷移図

Figure 2　Page flow diagram of our source code viewer.

Class Name	Score
Assert.java	0.000
AssertionFailedError.java	0.920
ComparisonCompactor.java	0.000
ComparisonFailure.java	0.085

図 3　クラス単位の可読性スコア表示

Figure 3　A page view of java classes and their readability scores.

AssertionFailedError.java

Score	Code
0.364	1 package junit.framework;
1.000	3 /** 4 * Thrown when an assertion failed. 5 */ 6 public class AssertionFailedError extends AssertionError {
0.997	8 private static final long serialVersionUID = 1L;
0.157	10 public AssertionFailedError() { 11 }

図 4　分割コードの可読性スコア表示

Figure 4　A page view of readability scores by divided partial statements.

4.2 可読性のスコア表示

(1) 表示方法

見やすさのため Buse らの可読性のスコアを小数点第 3 位まで表示し，そのスコアの良し悪しを直感的に判別できるようにスコアの背景を着色する（図 3，図 4）．スコアが 0 に近いほど読みにくいことを表すために赤に，高いほど読みやすいことを表すために緑で表示し，中間のスコアは赤と緑の中間色で表示する．

(2) 修正後のソースコードの配置とスコア表示

開発者は図 4 を見て可読性の低い部分を確認しソースコードの修正を試みる．開発者は自身が修正したソースコードのスコアがどの程度変化したか確認したいはずである．そこで，図 5 に示すように，修正するソースコードと同一のディレクトリ内に tmp ディレクトリを追加し，その中に修正するファイルをコピーしソースコードを修正する．tmp ディレクトリ内に同名のファイルが存在すれば可読性のスコアを並べて表示する．修正前後のスコアの表示の例を図 6 および図 7 に示す．図 6 と図 7 はそれぞれ図 3 と図 4 に対応している．

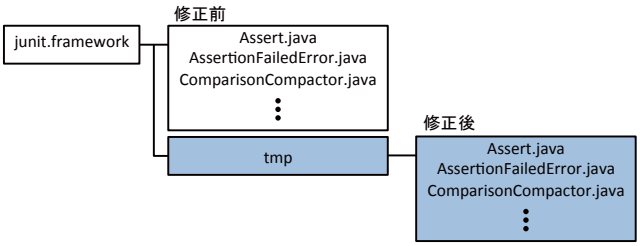

図 5 修正後コードのディレクトリ配置

Figure 5 The directory layout of modified source codes.

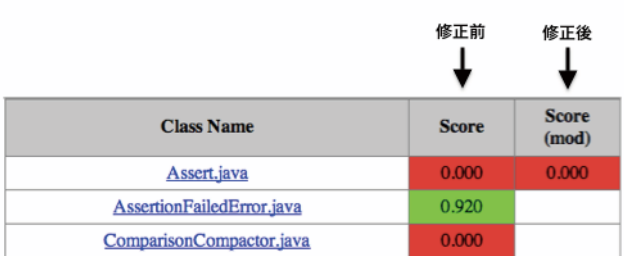

図 6 クラス単位の可読性スコア表示（修正後スコア併記）

Figure 6 A page view of java classes and their readability scores with modified class scores.

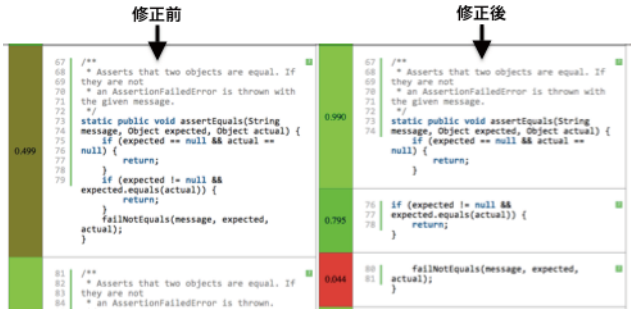

図 7 分割コードの可読性スコア表示（修正後スコア併記）

Figure 7 A page view of readability scores by divided partial statements with modified code scores.

5. 評価実験

5.1 被験者とソースコード

Java のコードを本学科 4 年生 5 名に与えて各自の基準で読みやすくなるように修正させる．被験者である 5 名の学生はいずれも 1 年次より Java に関する講義や演習を履修し半期あたり 1 コマ以上の演習を継続的に経験している．実験に使用した Java のコードは被験者らが講義や演習などで経験のある教科書に掲載されているサンプルコードを提供サイトよりダウンロードして使用した[9]．本来であればオープンソースのコードを使用すべきだが，被験者が未経験のコードが多く含まれていると，文法やクラスの利用方法などの理解に時間を要し，コードの修正時間を正確に測定できないと考えたためである．また，コードの読みやすさの基準は被験者によって差があると考えたため，ダウンロードしたサンプルコードを読みにくい状態に改変して使用した．具体的にはインデントと空行の削除をおこなった．なお，ダウンロードしたコードにはコメントは含まれていない．実験に用いたコードの行数を表 3 に示す．開発したソースコードビュアーの利用有無の差異を評価するため，実験を 2 回おこなう．被験者の学習効果による影響を低減するため，それぞれの実験においてほぼ同サイズで異なるクラスのコードを使用した．

表 3 評価実験に使用したコード行数

Table 3 Source lines of code of java programs using in the experiment.

条件	クラス名	行数	計
支援なし	Circle	33	119
	CircleState	19	
	GraphicFileFilter	29	
	SelectState	38	
支援あり	Rectangle	34	98
	RectState	19	
	Shape	14	
	ShapeDocument	31	

5.2 実験手順

一般的な開発者との乖離を低減するため，被験者 5 名に対して Java コーディング標準[10] を説明し，開発したソースコードビュアーの使い方についてトレーニングする．説明とトレーニングの時間は 15 分程度である．その後，1 人ずつ個室にてコードを修正する．Eclipse などの統合開発環境にはソースコードの自動レイアウト修正機能があるが，個人の読みやすさの基準を測定するためこれらの機能は利用せず一般的なテキストエディタを用いてコードを修正する．なお，実験に使用するコンピュータは Macbook Pro 17 インチでテキストエディアは Sublime Text 2 である．また，ソースコードビュアーの開発者が実験に同席し，被験者の様子を記録するとともに被験者が必要とした場合に補助する．これらの条件で以下の 2 つの実験をおこなう．実験間には 5 分程度の休憩時間を設ける．

実験 1：支援なし

ソースコードビュアー用いず表 3 の支援なしに含まれる 4 つのクラス対象として自由に選択して被験者の感覚で Java のコードを修正する．修正時間は最大 20 分間として被験者の完了宣言をもって終了できるものとする．

実験 2：支援あり

ソースコードビュアーを用いて表 3 の支援ありに含まれる 4 つのクラスを対象として自由に選択して可読性のスコアを参照しつつ Java のコードを修正する．修正時間は支援なしと同様である．

5.3 コード修正時間

2 つの実験におけるコードの修正時間を図 8 に示す．被験者 3 を除いて「支援なし」が「支援あり」より修正時間が短かった．被験者 1 と被験者 5 は「支援なし」では時間枠の半分程度で修正を完了したものの「支援あり」では時

間枠を活用して修正するという変化がみられた．これはコードの読みやすさが数値として見えることによって，個人としては十分に読みやすいと感じたコードに対しても，さらに可読性を改善しようとする行動を促した結果と考えられる．

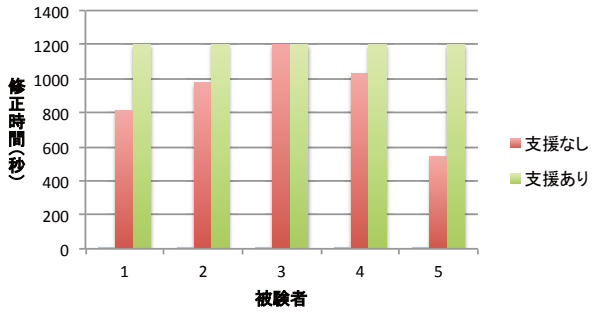

図 8 コード修正時間

Figure 8 The time to finish modifying the codes.

5.4 クラス単位の可読性のスコア変化

コード修正前後の可読性のスコアの変化量を表 4 に示す．なお，このスコアは図 3 に示したクラス単位の可読性スコアであり，その値をもとに変化量を計算した．すべての被験者およびすべてのクラスにおいてスコアが上昇した．GraphicFileFilter のように大幅に改善したクラスがある一方，Circle や Rectangle や SelectState のようにほとんど改善できないクラスもあった．SelectState では全被験者がともにスコアを全く改善できなかった．

表 4 被験者のコード修正による可読性スコアの変化量

Table 4 Buse *et al.* metrics score increments by subject's modification for source codes.

条件	クラス名	被験者 1	2	3	4	5	平均	σ
支援なし	Circle	0.012	0.099	0.011	0.002	0.003	0.144	0.236
	CircleState	0.014	0.244	0.007	0.002	0.003		
	GraphicFileFilter	0.684	0.649	0.591	0.325	0.241		
	SelectState	0.000	0.000	0.000	0.000	0.000		
支援あり	Rectangle	0.013	0.076	0.012	0.014	0.063	0.275	0.263
	RectState	0.024	0.150	0.349	0.236	0.491		
	Shape	0.278	0.983	0.293	0.454	0.695		
	ShapeDocument	0.147	0.219	0.105	0.258	0.635		

5.5 分割コードの可読性のスコア変化

コード修正前後の可読性のスコアの変化量を表 5 に示す．このスコアは図 4 に示した分割コードの可読性スコアをクラス単位で相加平均した値であり，その値をもとに変化量を計算した．分割コードの行数にはばらつきがあるが，Buse らのメトリクスは表 1 に示すようにソースコード特性の平均値あるいは最大値を利用しているため測定対象の行数に依存しない．このことから各クラスのスコアは分割コードのスコアの平均値でクラス全体のスコアを求めて支障がないと考えた．表 4 と表 5 は対応する同一のコードに対して測定した結果であるが，両者の平均値を比較すると表 5 の方が大きいことがわかる．被験者らの多くはクラス単位のスコアより分割コードのスコアを参照ながら修正作業をおこなっていたため，分割コードのスコアの方が修正結果を数値として適切にあらわしている．特に SelectState について比較すると，表 4 ではすべての被験者でスコアが 0 であったものの，表 5 ではすべての被験者の修正によってスコアが何かしら変化したことがわかる．

表 5 分割コードのスコアによる被験者のコード修正による可読性スコアの変化量

Table 5 Buse *et al.* metrics score increments calculated by divided source codes by subject's modification for source codes.

条件	クラス名	被験者 1	2	3	4	5	平均	σ
支援なし	Circle	0.019	0.099	0.374	0.416	0.012	0.186	0.225
	CircleState	0.012	0.287	0.005	0.002	0.003		
	GraphicFileFilter	0.592	0.796	0.316	0.165	0.116		
	SelectState	0.333	0.111	0.053	0.002	0.005		
支援あり	Rectangle	0.298	0.402	0.454	−0.051	0.690	0.447	0.243
	RectState	0.021	0.450	0.605	0.256	0.331		
	Shape	0.481	0.990	0.418	0.488	0.714		
	ShapeDocument	0.632	0.377	0.586	0.176	0.631		

5.6 有効性評価

(1) 統計的評価

被験者 5 名が修正したコードの可読性のスコアを調べたところ正規性および等分散性は認められなかった．また，2 つの実験において被験者は対応するもの実験に使用したコードが異なることから独立した 2 群と考え，支援なしおよび支援ありについて統計的な差をマン・ホイットニの U 検定によって調べた．結果として，クラス単位のスコアおよび分割コードのスコアにおいて $P < 0.01$ でそれぞれ差異があることが統計的に有為に示された．

(2) アンケート

実験終了直後にすべての被験者から以下の 3 項目について質問しメールで回答を得た．表 6 に質問文と整理した回答文を示す．表 6 の有効性欄は筆者が回答文を読んで有効と回答があったものに○を，有効性について判別できないものに△を，有効性に反対するものに×を記した．回答文からソースコードビュアーの表示方法および分割コードによるスコア表示はいずれも概ね有効であった．特に分割したコードについてスコアが表示されるため，可読性のスコアの低い部位を効率良く特定して修正作業することに役立ったようである．しかし，個々人の読みやすさの基準と可読性のスコアが必ずしも一致しないこと，数値化されることによって読みやすくするという本来の目的を忘れてしまう傾向があることがわかった．

表 6 実験直後の被験者によるアンケートの質問文と回答

Table 6 Questionnaire for the experiments and their responses

	質問文
Q1	可読性のスコアの表示方法は修正作業に有効だったでしょうか。
Q2	分割コードのスコア表示は修正作業に有効だったでしょうか。
自由	ソースコードビュアーの使用感について自由に回答してください。

設問	回答		
	被験者	有効性	コメント
Q1	1	○	緑になると安心した。逆にどうすれば赤を緑に変えられるか意識した。
	2	○	赤と緑の中間色ではなく明瞭な色分け（虹色等）の方がよい。
	3	○	スコアの絶対値ではなく増分量で色を変えた方がよい。
	4	○	色ではないがスコアが0.0001と極小の場合、修正効果が非常に小さいと感じた。
	5	○	色表示はこの方式で問題ない。
Q2	1	○	有効だと感じた。
	2	△	分割コードのスコアはあまり見なかった。
	3	○	有効だと感じた。
	4	○	修正が必要な部分を特定してプログラムを修正できるため有効と感じた。
	5	○	有効と感じた。ただ分割コードのスコアが上昇しても全体スコアが変化しないことに違和感を感じた。
自由	1		空行やコメントを入れるだけでスコアがあがるが本来の読みやすさとは違うと感じた。
	2		import文はどう修正してもスコアが上昇しないので対象から外すべき。
	3		分割しすぎると修正内容とスコアの対応がわかりにくくなる。
	4		スコアを再計算する操作が面倒だった。リアルタイムで再計算されるといい。
	5		空行を入れるとスコアが上がるが「本当に見やすいのか？」と思った。

6. 考察

6.1 可読性スコアの表示方法

評価実験の被験者の回答（Q1）から，ソースコードビュアーのスコアの表示方方法について概ね有効であることが確認できた．一方，被験者 1 のようにスコアや色に過度にこだわるケースや，被験者 4 のように色には表現できない変化量について軽視する傾向があることがわかった．修正対象であるコード片の可読性のスコアが低い場合，ソースコードの表層的特徴を単に整える程度の変更ではスコアが改善しないことを示している．本来のリファクタリングで主題となるプログラム構造の見直しなども含めた変更が必要と考える．今回の実験では限られた時間での修正作業であったため，インデントや空行やコメントの追加といった表層的な修正が主であった．したがって，可読性のスコアを改善するコードの修正方法やその戦略の分類や分析については今後の研究にて検証が必要である．一方，被験者 3 の回答で「スコアの絶対値ではなく変動量でスコアを着色してはどうか」といった意見があった．確かに被験者がコードを修正した時点でその修正内容に対する評価を得たいだろうし，その方が被験者の作業支援に貢献できる可能性があることも十分に考えられる．Buse らのメトリクスを実開発に適用する場合，スコアの絶対値をある基準以上にすべきなのか，改善した変化量を評価するべきか，さらに研究が必要である．

6.2 ソースコードの分割方法

評価実験の被験者の回答（Q2）から，空行による分割という単純な基準であるものの，概ね有効であることが確認できた．クラス単位のスコア（表 4）では， SelectState のようにすべての被験者において修正後のスコアの変化量が 0 となったクラスがあったものの，分割コードのスコア（表 5）では，すべての被験者において SelectState のスコアが非 0 になった．被験者にとってはコードを修正した結果で何かしらのスコアの変化を得たいはずであり，分割コードのスコアを示すことで被験者の作業支援に寄与したと考える．しかし，被験者 5 の回答のように，分割コードのスコアを改善してもクラス単位のスコアが変動せず困惑することがあった．被験者が分割コードにコメントを挿入するなどしてスコアを改善しても，クラス全体を対象として可読性のスコアを再計算した場合，追加したコメント行数が当該クラスの他のコード片に含まれるコメント行数で平均化されるため修正によるスコアの増加量が圧縮される場合がある．この問題を含む Buse のメトリクスに関する問題については Posnett ら[11]，Dorn ら[12]，Katzmarski ら[13]によって指摘されており，改善手法も提案されている．本稿の評価実験では Buse らのメトリクスの活用方法を検討することに主眼においたため，同一のコードに対してクラス単位と分割コードの 2 種類のスコアを被験者に提示した．被験者の修正内容とスコアをより強く対応させるためには，クラス単位のスコアと分割コードのスコアの差異はなくす必要があり，そのためには 5.5 で示した分割コードのスコアの相加平均をクラス単位のスコアとして採用する，あるいは他のメトリクスを採用するなどの改善方法を検討する必要がある．

また，アンケートへの回答（自由）から，被験者が修正作業の過程で空行やコメントを入れることで比較的簡単にスコアが上昇することを体験したようである．実験を補助しているときに注意したのだが，スコアを改善することに集中するあまりソースコードを読みやすくする本来の目的から逸脱して，本来は不要な空行等を挿入してスコアを上げようとした被験者も見られた．一般的に，比較的長いコードを意味ある単位にわけるために空行を使用するはずで，そのため空行をコード分割の区切り記号とした．この点については被験者の選定もしくは事前教育で対処できるのか，また一般的な開発者やオープンソースなどに適用して問題がないか確認する必要がある．

6.3 可読性改善過程における有効性

コード修正に要する時間に関しては，図 8 よりソースコードビュアーを用いた方がコード修正に要する作業時間が長くなった．作業効率の観点ではソースコードビュアーを用いることが悪影響していると考えられるが，個人の基準で十分に読みやすいと感じてもソースコードビュアーでスコアが赤く表示されるなどスコアが十分でないことを確認した場合，さらに修正を促す効果があったと考えられる．自然言語で書かれた文書の可読性基準の 1 つである Flesch-Kincaid Grade Level[14]のように，ソースコードに関して客観的な可読性の基準を作ることを Buse らは目標としている．本稿では簡単な評価実験ではあるが，Buse らのメトリクスがコードの可読性に関する客観的な評価基準として利用できる可能性を示唆していると考える．

可読性の改善に関しては，評価実験の結果からソースコードビュアーを用いた方がより大きくスコアを改善できることが統計的に示された．スコアを可視化することでスコアにこだわるあまり本来は不必要な修正もなされたが，その点も含めて可読性をメトリクスで判定できるように改善

することで対処できると考える．一方，Buse らのメトリクスはすべての Java のステートメントの並びについて可読性を評価できることは保証していない．例えば，被験者 2 の自由回答のように import 文が極端に低いスコアになることがわかっている．現状では Buse らのメトリクスをそのまま実開発に適用するには問題がある．可読性向上のための支援に適さないスコアについては非表示とするか，あるいは再学習して学習モデルを再構築する必要がある．

ソースコードの可読性を評価する方法としては，Hastead や McCabe らのメトリクスや CheckStyle[15]などの静的解析ツールにルールを定義することで評価すること可能なケースはある．しかし，Buse らのメトリクスの利点は，コード片とそのコード片に対する評価があれば，ルールを定義せずに可読性を測定できるところにある．可読性の評価には一般的な原則はあるとしても，組織やチームのスタイルはあるはずである．その場合，組織やチームのコード資産を学習データに用いることで，その組織やチームに適した可読性のメトリクスを構築することも可能であるため，Buse らのメトリクスを用いること，あるいは Buse らのメトリクスを活用した支援ツールを研究することには意義があると考えている．

7. おわりに

本稿では研究の初期段階として，Buse らが提案した可読性のメトリクスに着目し，開発者がソースコードの可読性を客観的に評価し，可読性の低い部位を効率的に特定する修正作業を支援するソースコードビュアーを提案した．評価実験により，スコアの表示方法およびソースコードの分割表示について概ね有効であることが確認できた．また，被験者 5 名によるコード修正の結果，ソースコードビュアーを利用した方がより大きく可読性のスコアを改善できることが統計的にも示すことができた．また，可読性のスコアを可視化することで，個人の基準では十分に読みやすいと感じてもスコアが十分でない場合，さらに修正を促す効果があった．一方，スコアを上げるために不必要な修正を加えるなどの問題もあった．今後の研究で明らかにしていきたい．

Buse らが提案したメトリクスは *ISSTA '08* にて発表されてから 5 年という期間が経過している[16]．コードの可読性の評価という古くから研究されてきたテーマに対して新しい視点を導入した．本稿で得られた知見より，コード修正作業に関して Buse らのメトリクスの有効性を示すことができたが，同時にいくつか問題があることがわかった．重要な懸念の 1 つとして，可読性スコアにしたがって修正したコードが本当に読みやすいか，がある．ソースコードビュアーのような支援ツールの研究とあわせて，より根本的な可読性の構造や現象の分析についても研究していきたい．

参考文献

1) B. Boehm and V. R. Basili, “Software defect reduction top 10 list,” Computer, vol. 34, no. 1, pp. 135-137, 2001.
2) L. E. Deimel Jr., “The uses of program reading,” SIGCSE Bull., vol. 17, no. 2, pp. 5–14, 1985.
3) Fowler, Martin, and Kent Beck, "Refactoring: improving the design of existing code," Addison-Wesley Professional, 1999.
4) Raymond P.L. Buse, Westley R. Weimer, "Learning a Metric for Code Readability," IEEE Transactions on Software Engineering, vol. 36, no. 4, pp. 546-558, July/August, 2010.
5) Maurice H. Halstead, “Elements of Software Science,” Elsevier Science Inc., 1977.
6) T. McCabe, “A Complexity Metrics”, IEEE Transactions on Software Engineering, vol.2, no.4, pp. 308-320, 1976.
7) jUnit.org: “jUnit: A programmer-oriented testing framework for Java”, available from (http://junit.org)
8) Readability Metric: Buse et al., available from (http://arrestedcomputing.com/readability/)
9) 高橋麻奈，やさしい Java オブジェクト指向編サンプルコード, avairable from (http://homepage3.nifty.com/~mana/yasao.html)
10) Code Conventions for the Java Programming Language: Oracle Corp., available from (http://www.oracle.com/technetwork/java/codeconv-138413.html)
11) D. Posnett, A. Hindle, and P. T. Devanbu, “A simpler model of software readability,” Working Conference on Mining Software Repositories, pp. 73–82, 2011.
12) Jonathan Dorn, “A General Software Readability Model”, MCS Thesis avairable from (http://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf)
13) Katzmarski, B.; Koschke, R., "Program complexity metrics and programmer opinions," Program Comprehension (ICPC), 2012 IEEE 20th International Conference on , pp.17-26, 11-13 June 2012
14) R. F. Flesch, “A new readability yardstick,” Journal of Applied Psychology, vol. 32, pp. 221-233, 1948.
15) Checkstyle, avairable from (http://checkstyle.sourceforge.net)
16) Raymond P.L. Buse, Westley R. Weimer, “A metric for software readability,” in International Symposium on Software Testing and Analysis, pp. 121-130, 2008