電子情報工学専攻

Advanced Electronic and Information Engineering Course

平成29年度 専攻科特別研究論文

ソフトウェアメトリクスを用いた高可読性コーディング能力の定量的評価

Quantitative Evaluation of Highly Readable Coding Ability Using Software Metrics

指導教員名　上野 秀剛

論文提出者名　中村 優太

独立行政法人 国立高等専門学校機構

奈良工業高等専門学校 専攻科

National Institute of Technology, Nara College

Faculty of Advanced Engineering

ソフトウェアメトリクスを用いた 高可読性コーディング能力の定量的評価

Quantitative Evaluation of Highly Readable Coding Ability Using Software Metrics

中村 優太
Yuta Nakamura

独立行政法人 国立高等専門学校機構
奈良工業高等専門学校 専攻科 電子情報工学専攻
大和郡山市矢田町 22 番地（〒 639-1080）
National Institute of Technology, Nara College, Faculty of Advanced Engineering
22 Yata-cho, Yamatokoriyama, Nara 639-1080, Japan

Abstract— On the software development site, the time cost of the task of reading the source code is large and it is required to be implemented with highly readable source code. However, at the educational site, readability is rarely evaluated, and few students implement implementations that are conscious of readability. In this research, we try to quantitatively measure the readability of the whole source code using software metrics. In this paper, we try quantitative evaluation method for differences in indentation and readability depending on difference in algorithm.

Keywords— Readability, Coding Ability, Software metrics, Conjoint analysis, Cyclomatic complexity

関連業績リスト

1. 中村 優太, 上野 秀剛, “ソフトウェアメトリクスを用いた高可読性コーディング能力の定量的評価,” 教育工学研究会（ET）, 2018-03-03 [予定].

目次

1. はじめに

ソフトウェアライフサイクルにおいて，保守作業量は 70 %と高い割合を占めると言われる [1]．さらに，保守作業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 [2]．可読性の高いソースコードで開発することはソフトウェア開発の生産性を高め，品質の向上にも寄与すると考えられる．

しかし，大学等のプログラミング教育の現場において，プログラムが正しく動作するか（以下，可動性と呼ぶ）は評価されるものの，そのソースコードの可読性までは評価されないことが多い．原因として，可読性は定量的な評価基準を設定しにくい要素であるからと考える．そこで本研究では，ソースコードの可読性をソフトウェアメトリクスを用いて定量的に計測する手法を提案する．

可読性に影響を与える要素として変数名や改行，コメント，インデント，アルゴリズムなどが考えられる．中でも本研究では，インデントとアルゴリズムによる可読性を評価する手法を提案する．

大学などのプログラミング教育現場では，プログラミング初学者は多いと考えられ，初学者はインデントを整えてコーディングしない場合が多いと考えられる．インデントは視覚的に可読性を高める効果があり，アルゴリズムは論理構造を理解する要素として大きく可読性に影響すると考えられる．またアルゴリズムにおいても可読性を評価されないためか，可動性のみを重要視した読みにくいコードで実装する．

本研究では，インデントとアルゴリズムによる可読性を評価するメトリクスを提案する．

可読性を定量的に評価し，フィードバックを行うことで，学習者は可読性を意識したコーディングを習得することができると考えている．

2. 関連研究

2.1 プログラミング能力の定量化に関する研究

プログラミング能力の定量的評価方法に関する既存研究としては，プログラマのデバッグ能力に着目したものがある．

松本ら [3] は，各エラーが作成されてから除去されるまでの時間的効率であるエラー寿命に着目している．この研究では，能力の高いプログラマほどエラーを作り込まず，エラーが含まれたとしても短い期間でそれを取り除くことができるという考えに基づき，プログラミング能力を定式化している．

また，高田ら [4] は，プログラマのデバッグ能力をキーストロークから測定する方法を提案している．プログラマのデバッグ中の状態を「コンパイル」,「プログラム実行」,「プログラム変更」の 3 種類に分類した上で，デバッグ中はこの 3 状態の繰り返しであるとし，これらの 3 状態をキーストロークから検出する．デバッグ能力の高いプログラマは 3 状態の少ない繰り返しで，バグを取り除くという特徴からデバッグ能力の定量化している．いずれの方法もデバッグ作業中のプログラミング能力を定量的に評価している．

2.2 ソースコード可読性に関する研究

ソースコードの可読性評価に関する研究は Buse らの可読性メトリクスがある．Java のソースコードの断片を 100 個集め，それらに対して学生 120 人による可読性評価を行った．次に，ソースコードの可読性は { 変数の長さ，インデントの数，1 行の長さ，数値の数，変数の数，予約後の数，変数の種類 } の平均値・最大値および { 空行，空白，コメント行，代入文，分岐，ループ，算術演算，比較，括弧，ピリオド } の数の平均値に重み付けをした線形和で表せると仮定した．主観評価を用いた機械学習によりこれらの特徴量の重みを決定して定量化を行った．Buse らの研究では，可読性に関わる多くの要素を機械

学習を用いて 1 つのメトリクスで評価している．

本研究では，教育現場で学生へ可読性をフィードバックする方法として，1 つ 1 つの要素に対する可読性をフィードバックすることで，学生はよりその要素を意識したコーディングを身につけることができると考え，インデントとアルゴリズムによる可読性を定量化する．

3. 視覚的可読性評価手法

3.1 インデントと可読性

インデントはプログラムの構造を視覚的に明確に表す手法として用いられている．インデントはネスト構造の範囲を視覚的に表すことで可読性に大きな影響を与えていると考える．そこで本章ではソースコードのおけるインデントによる可読性を定量的に評価するメトリクスを提案し，それを用いた高可読性コーディング能力の評価方法を提案する．

3.2 提案メトリクス

本節では，インデントによる可読性を定量的に表すメトリクスを提案する．提案するメトリクスでは，ソースコードのインデント情報を，ネスト構造 (3 種類) とインデントの状態 (3 種類) という観点から 9 種類に分類し，その 9 種類がそれぞれソースコードの可読性に与える重みを求める．求めた重みを基にソースコードのインデントによる可読性を定量的に表す．

3.2.1 インデント状態

本研究では，インデントの状態を 3 種類に分類して考える．インデントの幅を半角スペース 4 個分にして，ある if 文をコーディングした例を図 3.1 に示す．図 3.1(a) では 1 行目にある if 文の内部が 7 行目まで続いていることをインデントで表している．提案するメトリクスではこの状態をインデント状態「あり」と評価する．3.1(b) は if 文内部を表すインデントがなされていない．この状態をインデント状態「なし」と評価する．また図 3.1(c) はインデントがされてる行と，されていない行が混じっている．この状態を インデント状態「混合」と評価する．

インデント「あり」はインデントを正しくできており，ネスト構造が視覚的に理解でき

```
1: if(a < b){
2:     System.out.print(”HelloWorld1”);
3:     System.out.print(”HelloWorld2”);
4:     System.out.print(”HelloWorld3”);
5:     System.out.print(”HelloWorld4”);
6:     System.out.print(”HelloWorld5”);
7: }
```

(a) インデントあり

```
1: if(a < b){
2: System.out.print(”HelloWorld1”);
3: System.out.print(”HelloWorld2”);
4: System.out.print(”HelloWorld3”);
5: System.out.print(”HelloWorld4”);
6: System.out.print(”HelloWorld5”);
7: }
```

(b) インデントなし

```
1: if(a < b){
2: System.out.print(”HelloWorld1”);
3:     System.out.print(”HelloWorld2”);
4:     System.out.print(”HelloWorld3”);
5: System.out.print(”HelloWorld4”);
6:     System.out.print(”HelloWorld5”);
7: }
```

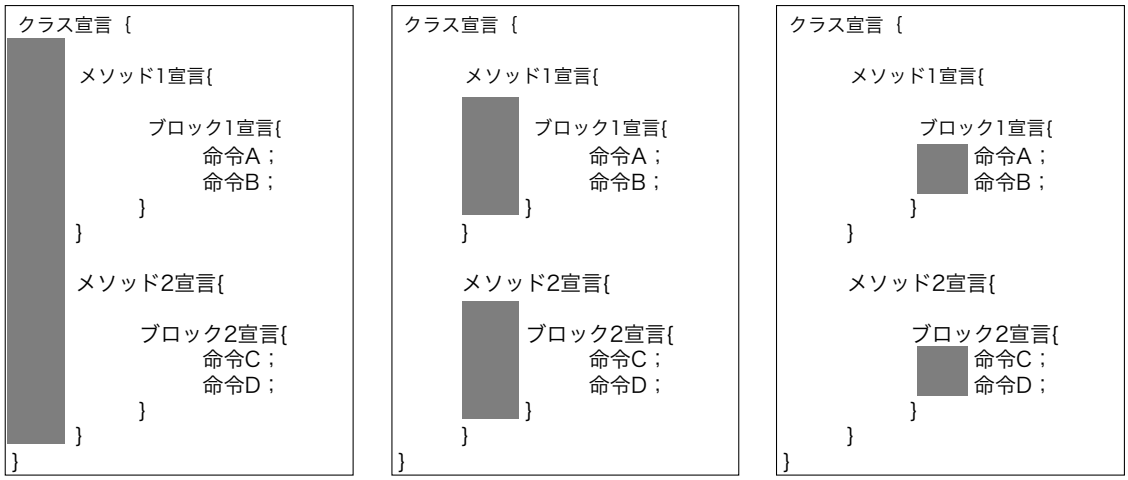
(c) インデント混合

図 3.1: インデント状態

るインデントの使い方である．一方インデント「なし」は，ネスト構造が発生している場面において，それが視覚的にわかる状態ではなくインデント「あり」に比べて可読性は低いと考えられる．また，インデント「混合」のソースコードを読む時，読み手は 1 行 1 行どのネストの文なのかを判断しながら読まなければならず，混乱を生む．そのため，インデント「あり」「なし」に比べて，可読性がより低いと考える．本提案メトリクスでは，この 3 種類の状態による可読性の違いを定量化する．

3.2.2 ネスト構造

インデントはネスト構造の範囲を視覚的に表す．プログラムにおけるネスト構造とは，クラス，メソッド，ブロック (if 文や for 文など) の 3 種類で存在する．ソースコードにおける，それぞれのネスト構造を表すインデントの領域を図 3.2 に示す．図 3.2(a) に示すクラスのインデント領域では，グレーの領域をタブやスペースなどを用いて空白にすることで，クラス宣言後，その内部がソースコードのどこまで続いているのかを表す．同様に図 3.2(b) のメソッド，図 3.2(c) のブロックのインデント領域も同様に，それぞれの内部がソースコードのどこまで続くのかを表している．この領域によりそれぞれのネスト構造を視覚的に表している．



(a) クラス　　(b) メソッド　　(c) ブロック

図 3.2: ソースコードのネスト構造とインデント領域

3.2.3 インデント情報

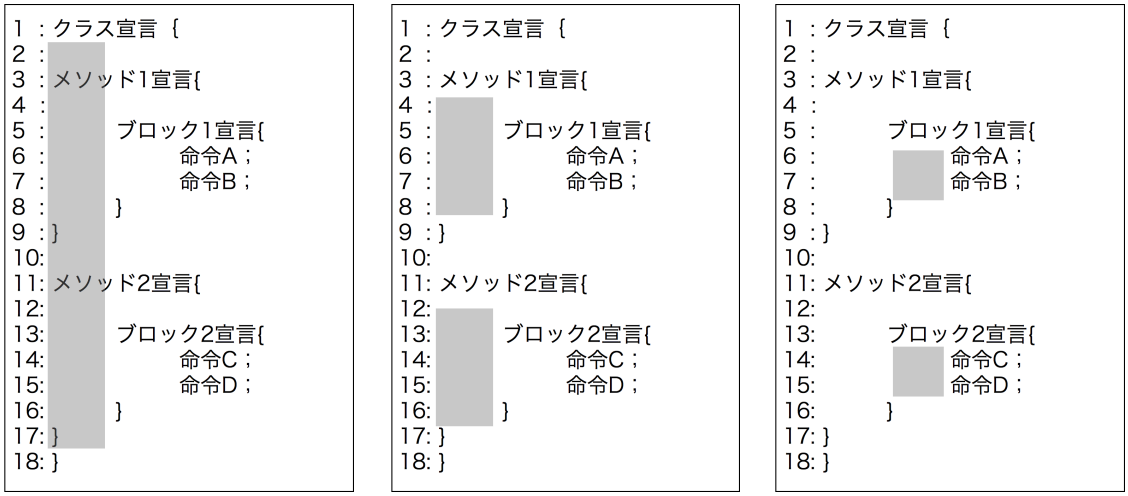
本研究では，ソースコードのインデント情報を各ネスト構造 3 種類 (クラス，メソッド，ブロック) の各状態 3 種 (あり，なし，混合) の 9 種類とする．ネスト構造とインデント状態の関係を表 3.1 に示す．1 つのソースコードのインデント情報はクラスインデント (あり or なし or 混合)，メソッドインデント (あり or なし or 混合)，ブロックインデント (あ

り or なし or 混合) の 3 状態から表される．(以後，ソースコードのインデント情報を「インデント情報 { クラスの状態，メソッドの状態，ブロックの状態 }」で表す．例えば，クラスインデント「あり」，メソッドインデント「なし」，ブロックインデント「混合」の場合，インデント情報 { あり, なし, 混合 } と表す．)

表 3.1: インデント情報

ネスト構造	状態
クラス	あり なし 混合
メソッド	あり なし 混合
ブロック	あり なし 混合

なお，ネスト構造を宣言した位置から空白が空いているかどうかで，インデントがなされているか，なされていないかを判定する．例として，図 3.3 にインデント情報 { なし, あり, あり } のソースコードを示す．図 3.3(a) に示されているクラスのネスト構造を表す領域には，3 行目と 11 行目がメソッド宣言のインデントがなされておらず，クラスインデントは「なし」である．クラスインデントが正しく「あり」である場合，2 行目から 17 行目のコードは 1 つインデントが多くされている状態となる．そのため，2 行目から 17 行目のインデントは本来の位置からずれてしまい，正しいインデントの位置ではなくなる．しかし，ネスト構造を宣言した位置から空白が空いているかどうかでインデントを判定する．今回の例では，3 行目と 11 行目でメソッドが宣言されてから，それぞれ 9 行目と 17 行目のメソッドの終わりまで空白があるため，メソッドインデント「あり」と判定する．メソッドを読む場合には，メソッドの宣言からメソッドの終わりのメソッド領域がインデントによりわかるためである．ブロックも同様に宣言後，そのブロックの終わりまで空白が空いているため，インデント「あり」と判定する．



(a) クラスインデント (なし)　(b) メソッドインデント (あり)　(c) ブロックインデント (あり)

図 3.3: ソースコード情報 { なし，あり，あり }

3.2.4　重みの算出

提案メトリクスでは，表 3.1 の 9 状態が可読性に与える重みを求める．本研究では，重みの算出にコンジョイント分析 [6] を用いる．コンジョイント分析はマーケティング等の分野で広く一般的になった手法である．ある事物や事象に対して，被験者がどのような要素を重視するかを知りたい場合に用いる．

テレビの購入の例を挙げる．消費者はテレビを購入するとき，画面サイズ，解像度，メーカー，価格の 4 つのどの要素を重視するのか，また画面サイズには “32 インチ”，“42 インチ”，価格は “50,000 円”，“90,000 円” といった選択肢があるとき，どの選択肢がどれだけいいのかを求めたい．そのときコンジョイント分析では，消費者に対してテレビの特性を組み合わせた図 3.4 のようなコンジョイントカードを複数提示して，コンジョイントカードに対して順位づけを行ってもらう．

コンジョイント分析は被験者がコンジョイントカードにつけた順位を基に，(1) 部分効用値の算出 (どの水準が，どの程度重視されているかを求める)，(2) 重要度の算出 (どの属性が，どの程度重視されているかを求める)，(3) 全体効用値の算出 (各コンジョイントカードの総合得点を求める) によって行われる．具体的には，

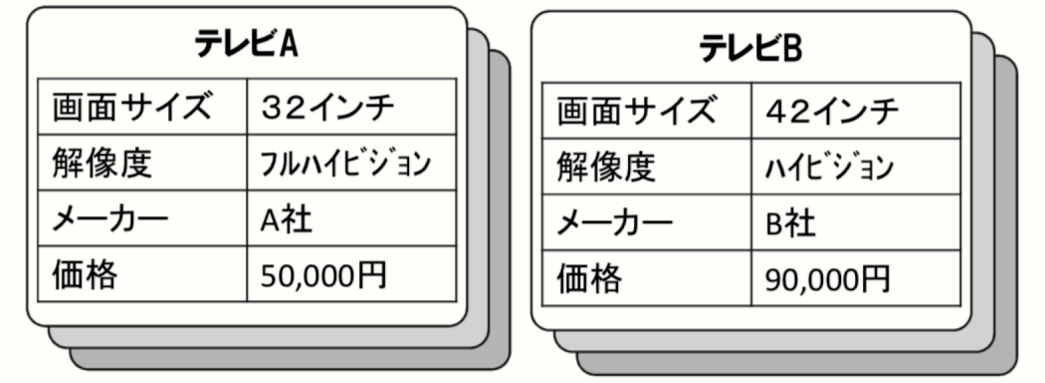


図 3.4: コンジョイントカード (テレビ)

(a) 被験者がコンジョイントカードを順位付けした結果について，重回帰分析を行う．

(b) 水準ごとにデータ件数と偏回帰係数を掛算し，データの総数で割ることによって，加重平均を求める．

(c) 偏回帰係数から加重平均を引いて，部分効用値 (重み) を求める．

(d) 属性ごとに，部分効用値の最大値と最小値の差 (レンジ) を求める．

(e) 各属性のレンジの合計に占める割合 (重要度) を求める．

(f) コンジョイントカードの水準を各部分効用値に置き換え，これらの値を加算した値 (全体効用値) を求める．

つまり，コンジョイント分析は，“画面サイズ”や“価格”といった要素 (属性という) と，“32 インチ”，“42 インチ”や“50,000 円”，“90,000 円”といった選択肢 (水準という) の組み合わせとその順位付けによって，複数の尺度間で，何が，どのくらい重視されているのかを定量的に把握することができる手法である．

本研究の重み (部分効用値) の算出では，可読性について順位をつけ，コンジョイント分析を用いることで，“クラス”や“メソッド”といった属性と，“あり”，“なし”，“混合”といった水準の組み合わせとその順位付けによって，被験者は可読性を評価する上でどの属性や水準をどのくらい重視されているのかを定量的に把握することができる．また，全体効用値によって，コンジョイントカードを定量的に評価できるようになる．この全体効用値をソースコードのインデントによる可読性を定量的に表すメトリクス値として提案する．全体効用値は，それぞれのコンジョイントカードを総合評価したものであり，可読性

について順位をつけたものをコンジョイント分析した結果から得られる全体効用値はソースコードの可読性を表すと考えることができる．

3 属性 (クラス，メソッド，ブロック) のそれぞれ 3 水準 (あり，なし，混合) を用いて，表 3.2 に示す提案メトリクスで用いるコンジョイントカードの組み合わせを作成する．実際にコンジョイント分析を行うため，3.3 節に示す被験者実験を行い，全体効用値を求める．

3.3 被験者実験

コンジョイント分析を行うためのコンジョイントカードに対する順位データを収集するために被験者実験を行う．実験では被験者に表 3.2 のコンジョイントカードに基づいたインデント情報が異なるソースコードを読んでもらい，可読性について順位をつけてもらう．その順位データをコンジョイント分析し，部分効用値，重要度，全体効用値を求める．

3.3.1 コンジョイントカード数の削減

表 3.2 のコンジョイントカードは 27 パターンあり，27 個のソースコードを被験者に読んでもらうのは，負担が大きい．コンジョイント分析では，直交表を使用してをコンジョイントカードを減らして実験を行うことができる．そこで表 3.3 に示す $L_9(3^3)$ 型直交表に基づいて実験対象とするコンジョイントカードを削減する．この直交表は 3 属性，3 水準から作られるコンジョイントカードの組み合わせを 9 パターンに絞るためのものである．

$L_9(3^3)$ 型直交表に基づいてパターン数を削減した実験用コンジョイントカードを表 3.4 に示す．直交表の属性 1 には「クラスインデント」，属性 2 には「メソッドインデント」，属性 3 には「ブロックインデント」を割り当てる．また水準についても，1:「あり」，2:「なし」，3:「混合」を割り当てて実験用コンジョイントカードを作成する．

3.3.2 実験環境

被験者は奈良高専，情報工学科 3 年から 5 年，もしくは情報工学科出身の電子情報工学専攻に所属する 17 名を対象に行う．被験者実験はを 5 名もしくは 6 名に対して同時に行う．被験者に対して，表 3.4 のコンジョイントカードに示されたインデント情報を Java

表 3.2: コンジョイントカードの組み合わせ

card No.	クラス	メソッド	ブロック	全体効用値
1	あり	あり	あり	
2	あり	あり	なし	
3	あり	あり	混合	
4	あり	なし	あり	
5	あり	なし	なし	
6	あり	なし	混合	
7	あり	混合	あり	
8	あり	混合	なし	
9	あり	混合	混合	
10	なし	あり	あり	
11	なし	あり	なし	
12	なし	あり	混合	
13	なし	なし	あり	
14	なし	なし	なし	
15	なし	なし	混合	
16	なし	混合	あり	
17	なし	混合	なし	
18	なし	混合	混合	
19	混合	あり	あり	
20	混合	あり	なし	
21	混合	あり	混合	
22	混合	なし	あり	
23	混合	なし	なし	
24	混合	なし	混合	
25	混合	混合	あり	
26	混合	混合	なし	
27	混合	混合	混合	

表 3.3: $L_9(3^3)$ 直交表

	属性 1	属性 2	属性 3
カード 1	1	1	1
カード 2	2	2	1
カード 3	3	3	1
カード 4	2	1	2
カード 5	3	2	2
カード 6	1	3	2
カード 7	3	1	3
カード 8	1	2	3
カード 9	2	3	3

表 3.4: 実験用コンジョイントカード

	クラス	メソッド	ブロック
カード 1	あり	あり	あり
カード 2	なし	なし	あり
カード 3	混合	混合	あり
カード 4	なし	あり	なし
カード 5	混合	なし	なし
カード 6	あり	混合	なし
カード 7	混合	あり	混合
カード 8	あり	なし	混合
カード 9	なし	混合	混合

で実装したものを 9 つ提示する．提示した 9 つのソースコードは同じ仕様，同じアルゴリズム，同じ構文を使っている．プログラムの仕様は「5 人分の国語と数学と英語の成績を入力し，最後に各教科の平均点を表示する」ものである．9 つのソースコードで異なるのは，インデントの付け方のみである．掲示方法は 1 枚の紙に 1 つのプログラムを印刷したものを 9 枚渡す．例として，図 3.5 にカード 1 インデント情報 { あり, あり, あり } のソースコードを示す．

被験者は制限時間 16 分間で 9 枚のソースコードに対して，可読性が高いと感じる順番に順位をつける．制限時間終了後，被験者に結果集計用紙を配布し，結果を記入してもらう．また，同時にアンケート用紙を配布し，アンケートにも回答してもらう．アンケートの内容，使い道については 3.4 節の結果と考察で述べる．

得られた順位データをコンジョイント分析する．

3.4　結果と考察

被験者 17 名に対して，順位データを収集し，コンジョイント分析を行った結果を表 3.5 に部分効用値，表 3.6 に重要度を示す．また部分効用値から求めた全体効用値を表 3.7 に示す．部分効用値からわかるように，被験者は各ネスト構造においてインデント状態「あり」が最も可読性が高いと感じ，インデント状態「混合」が最も可読性が低いと感じている．

重要度は被験者がどのネスト構造におけるインデントを重視しているかを表す度合いである．被験者はクラスインデント，メソッドインデント，ブロックインデントの順で重要であると感じていることがわかる．

全体効用値によりソースコードのインデント情報 27 パターンに対して，可読性が定量的にわかる．実際に教育現場などでソースコードを評価する場合には，全体効用値は数値としてわかりにくいため，全体効用値の最大値を 100 点，最小値を 0 点として正規化を行う．

この結果により，教育現場でソースコードのインデントによる可読性ををインデント情報から定量的に評価することができる．

```
import java.util.Scanner;

public class Main {
	static Scanner stdIn = new Scanner(System.in);
	static int[] kokuScore = new int[5];
	static int[] suScore = new int[5];
	static int[] eiScore = new int[5];

	public static void input(){
		System.out.println("5人分の成績を入力");

		for(int i=0; i<5; i++){
			System.out.println(i+1 + "人目");
			System.out.print("国語：");
			kokuScore[i]= stdIn.nextInt();
			System.out.print("数学：");
			suScore[i]= stdIn.nextInt();
			System.out.print("英語：");
			eiScore[i]= stdIn.nextInt();
		}

		System.out.println("5人分の入力が終了しました");
	}

	private static void output() {
		System.out.println("5人の平均点を表示");
		int kokuSum = 0, suSum = 0, eiSum = 0;

		for(int i=0; i<5; i++){
			kokuSum = kokuSum + kokuScore[i];
			suSum = suSum + suScore[i];
			eiSum = eiSum + eiScore[i];
		}

		System.out.println("国語：" + kokuSum/5.0);
		System.out.println("数学：" + suSum/5.0);
		System.out.println("英語：" + eiSum/5.0);
	}

	public static void main(String[] args) {
		input();
		output();
	}

}
```

青

図 3.5: 実験用ソースコード (インデント情報 { あり, あり, あり })

表 3.5: 部分効用値 (17 名分)

ネスト構造	状態	部分効用値
クラス	あり	0.96
	なし	0.31
	混合	-1.27
メソッド	あり	1.51
	なし	0
	混合	-1.51
ブロック	あり	1.39
	なし	0.18
	混合	-1.57

表 3.6: 重要度 (17 名分)

ネスト構造 (属性)	重要度 [%]
クラス	36.09
メソッド	34.66
ブロック	29.24

3.4.1 プログラミングレベル別分析

開発現場において，プログラマのレベルは上級者であると考えると，上級者と下級者でソースコードを読むときに重視するポイントが異なるとき，上級者が感じる可読性を定量的に評価することで，より上級者向けなコーディングを習得することができると考えた．そこで今回の実験では，アンケートによって被験者のレベルを上級者，中級者，下級者の 3 段階に分け，レベル別での分析を行った．被験者は学生のみであるが，プログラマというくくりで見ると，下級者から上級者への過程にあると考え，学生の上級者は下級者より開発現場にいる上級者に近い尺度で可読性を感じると思われ，学生のみでも上級者と下級者に分けて分析し，重視するポイントを見ることは優位であると考える．

本研究で，プログラミングレベルを以下の基準で下級者，中級者，上級者に分ける．被験者は奈良高専の学生であるため，下級者は「授業を受けていれば達成する項目」，中級

表 3.7: 全体効用値 (17 名分)

card No.	クラス	メソッド	ブロック	全体効用値	100 点換算
1	あり	あり	あり	3.86	100
2	あり	あり	なし	2.65	85
3	あり	あり	混合	0.9	64
4	あり	なし	あり	2.35	82
5	あり	なし	なし	1.14	67
6	あり	なし	混合	-0.61	46
7	あり	混合	あり	0.84	63
8	あり	混合	なし	-0.37	48
9	あり	混合	混合	-2.12	27
10	なし	あり	あり	3.22	92
11	なし	あり	なし	2.00	77
12	なし	あり	混合	0.25	56
13	なし	なし	あり	1.71	74
14	なし	なし	なし	0.49	59
15	なし	なし	混合	-1.25	38
16	なし	混合	あり	0.20	55
17	なし	混合	なし	-1.02	41
18	なし	混合	混合	-2.76	19
19	混合	あり	あり	1.63	73
20	混合	あり	なし	0.41	58
21	混合	あり	混合	-1.33	37
22	混合	なし	あり	0.12	54
23	混合	なし	なし	-1.10	40
24	混合	なし	混合	-2.84	18
25	混合	混合	あり	-1.39	36
26	混合	混合	なし	-2.61	21
27	混合	混合	混合	-4.35	0

表 3.8: レベル別重要度

	下級者	中級者	上級者
クラス	32.89	22.23	41.38
メソッド	32.44	35.96	33.97
ブロック	34.67	41.81	24.65

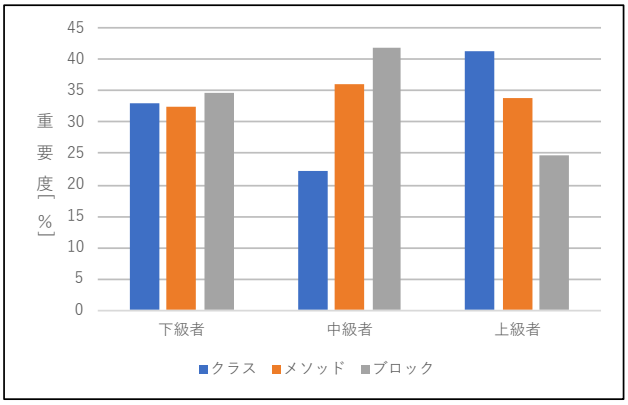


図 3.6: レベル別重要度

者は「授業の課題でより高度なコードを書いた者が達成する項目」,「授業の課題などでは実装するはなく，自らの勉強や課外で経験がないと達成できない項目」の基準でプログラミングレベルを分けた．

(下級者) 複数クラス&複数メソッドを用いたコードを書いたことがある．

(中級者) 複数クラス&複数メソッドを用いたコードかつ 500 行　999 行のコードを書いたことがある．

(上級者) 複数クラス&複数メソッドを用いたコードかつ 1000 行以上のコードを書いたことがある．

レベル別にコンジョイント分析を行った重要度の結果を表 3.8 と図 3.6 に示す．

図 3.6 から下級者はクラス，メソッド，ブロックの 3 つの属性に対してほぼ同等の重要度と認識していることがわかる．また中級者はブロックインデントの状態を最も重視し，メソッドインデント，クラスインデントの順に重視していることがわかる．上級者は中級者とは逆に，クラスインデントの状態を最も重視しており，メソッドインデント，ブロックインデントの順に重視している．下級者はどのインデントを重視すべきかわかっておらず，重要度がほぼ均等になったと考えられる．中級者はソースコードをブロックを理解し，メソッドを理解し，そしてクラスを理解するというボトムアップな読み方をすると考えられ，そのためブロックを読む際のインデントを重視すると考えられる．上級者は先にクラス内のメソッドの位置など全体の構成を先に読むことが言われている [7]．上級者は

中級者とは逆にトップダウンに読んで行くと考えられる．そのため，クラスインデントを最も重視し，メソッドインデント，ブロックインデントの順に重視するといった結果になったと考えられる．

下級者，中級者，上級者では重視するネスト構造が違うことがわかる．そこで，上級者が感じる可読性を定量的に評価するためには，上級者からデータを収集して，コンジョイント分析を行った結果をメトリクスとして定量化することが良い．今後の課題として，開発現場などの上級者からデータを収集して，定量化を行う必要があると考える．

4. 論理構造的可読性評価手法

4.1 論理構造と可読性

ある仕様に対して複数のプログラマがコーディングを行う場合，プログラマによって実装するアルゴリズムが異なる可能性がある．実装するアルゴリズムが異なると，ソースコードの可読性は異なってくると考えられる．

例として，3 つの数から最大値と最小値を求めて表示するプログラムを 2 パターンのアルゴリズムで実装したソースコードを図 4.1 に示す．アルゴリズム 1 は，「3 つの数を配列に格納し，for 文を用いて最大値と最小値を更新していき最終的に max と min に格納されている値が最大値と最小値になっている」アルゴリズムである．アルゴリズム 2 は，「3 つの数を変数 (a, b, c) に格納し，a, b , c の大小関係を if 文を用いて場合分けしていき，それぞれの条件にあった最大値と最小値を max と min に格納する」アルゴリズムである．

アルゴリズム 1 とアルゴリズム 2 では，アルゴリズム 1 が読みやすく，可読性が高いと感じる．このように同じ仕様のプログラムだが，アルゴリズムの違いによって可読性が異なる．

そこで本章ではアルゴリズムによる可読性を定量的に評価する手法を提案する．

4.2 提案手法

提案手法はメソッドの複雑さを表すメトリクスである McCabe のサイクロマティック数を改良することで，アルゴリズムの複雑さに起因する可読性を数値で表す．

```
1 : int[]n={1,2,3};
2 : int max=n[0];
3 : int min=n[0];
4 :
5 : for(int i=0; i<3; i++){
6 :     if(max < n[i]){
7 :         max = n[i];
8 :     }
9 :     if(min>n[i]){
10:         min=n[i];
11:     }
12: }
13: System.out.println("max = "+ max);
14: System.out.println("min = "+ min);
```

(a) アルゴリズム 1

```
1 : int a=1;
2 : int b=2;
3 : int c=3;
4 : int max,min;
5 :
6 : if(a>b){
7 :     if(c>a){
8 :         max=c;
9 :         min=b;
10:     }
11:     else{
12:         max=a;
13:         if(c<b){
14:             min=c;
15:         }
16:         else{
17:             min=b;
18:         }
19:     }
20: }
```

```
21: else{
22:     if(c>b){
23:         max=c;
24:         min=a;
25:     }
26:     else{
27:         max=b;
28:         if(c<a){
29:             min=c;
30:         }
31:         else{
32:             min=a;
33:         }
34:     }
35: }
36: System.out.println("max = "+max);
37: System.out.println("min = "+min);
```

(b) アルゴリズム 2

図 4.1: 最大値，最小値を求めるプログラム

4.2.1 サイクロマティック数

ソースコードの複雑度を表すメトリクスに McCabe のサイクロマチック数 [5] がある．サイクロマティック数は 1 つのメソッドの制御パス数を表す複雑度メトリクスであり，以下の式 4.1 で制御フローのリンク数とノード数から表される．

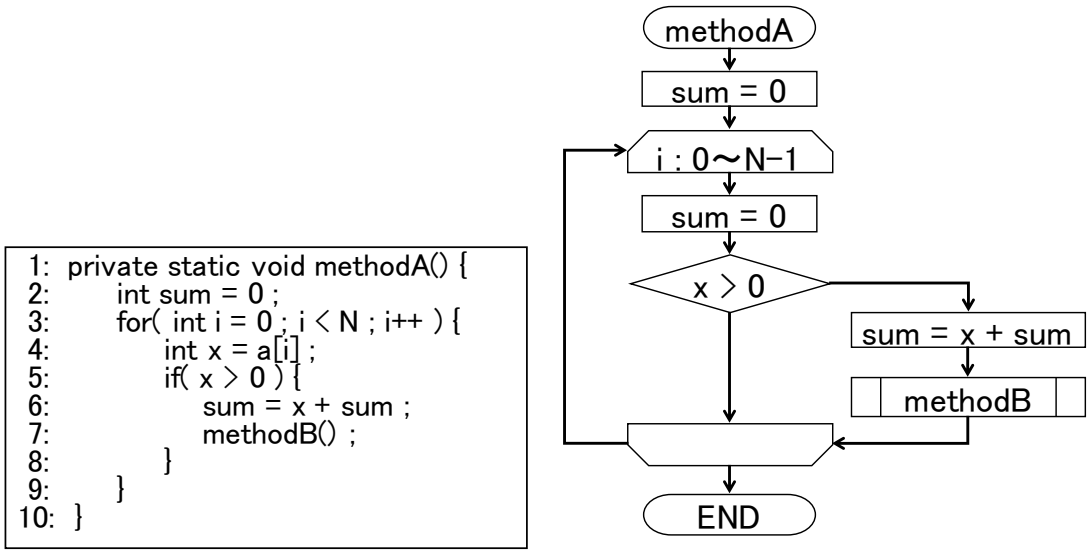
$$\text{サイクロマティック数} = (\text{リンク数}) - (\text{ノード数}) + 2 \tag{4.1}$$

例として，図 4.2 にあるメソッド A のソースコードとその制御フローを示す．

ノードは制御フローにおける命令の書かれた 1 つの箱を表し，リンクとはノードとノードをつなぐ矢印のことである．図 4.2(b) の例では，ノード数は 9，リンク数は 10 であるため，以下の式 4.2 よりメソッド A のサイクロマティック数は 3 となる．

$$(\text{サイクロマティック数})3 = (\text{リンク数})10 - (\text{ノード数})9 + 2 \tag{4.2}$$

直感的にはサイクロマティック数は「制御フローにおける閉じたループの数 + 1」で表すことができる．サイクロマティック数は，分岐やループが多いほどメソッドの複雑度が増すという考え方の元にある．メソッドが複雑なほどバグの混入リスクや発見されたバグの修正にかかる時間が増え，修正で新たなバグが混入する可能性も増えるため，再利用が困難になる．そのためソフトウェア開発現場では，メソッドごとのサイクロマティック数



(a) ソースコード　　(b) 制御フロー

図 4.2: ソースコードと制御フロー

を計測し，低く保つことで，保守性，移植性を維持する．また，サイクロマティック数はメソッドの複雑さを表す指標であるため，可読性にも影響があると考えられる．

アルゴリズムの違いによる可読性を評価するためには，ソースコード全体を可読性評価する必要があると考えられる．しかし，サイクロマティック数はメソッドごとの複雑さを評価するものであって，ソースコード全体の複雑さは表さない．そのため，アルゴリズムがどれだけ複雑であっても，1 つ 1 つのメソッドのサイクロマティック数が小さければプログラム全体が複雑だとは言えない．

そこで本研究ではサイクロマティック数を用いてアルゴリズムの違いによる可読性を定量的な評価方法を提案する．4.2.2 節で，サイクロマティック数を足し合わせることでソースコード全体の複雑さを表すメトリクスを示す．しかし，サイクロマティック数を足し合わせる方法は必ずしもアルゴリズムの複雑さを表すとは限らないという問題点がある．その問題点を解決した上で，アルゴリズムの複雑さを表すメトリクスとして，4.2.3 節で拡張サイクロティック数を提案する．

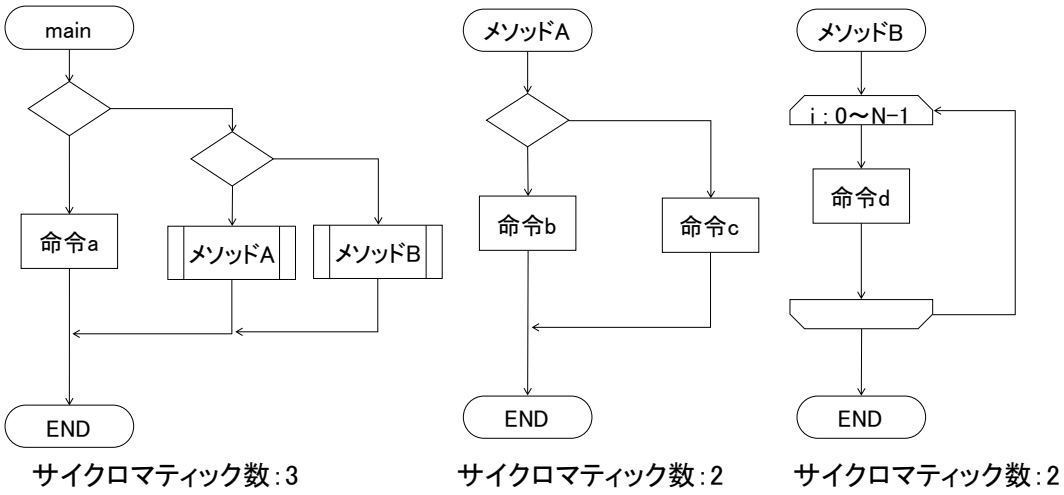


図 4.3: 制御フロー (メソッド 3 つ)

4.2.2 合計サイクロマティック数

ソースコード全体の複雑さを評価する方法として，全てのメソッドのサイクロマティック数を足しあわせる方法（以下，合計サイクロマティック数）が考えられる．例として，図 4.3 の制御フローのようにメインメソッドと 2 つのメソッドからなるプログラムを考える．

この例では，以下の式 4.3 より，合計サイクロマティック数は 7 となる．

$$(\text{合計サイクロマティック数})7 = (main)3 + (\text{メソッド } A)2 + (\text{メソッド } B)2 \quad (4.3)$$

また，同じアルゴリズムでメソッドの分割方法を変えたプログラムの制御フローを図 4.4 に示す．この例では，図 4.3 の例でメインプログラムから呼び出されていたメソッド A とメソッド B の呼び出し，またそれを判断する分岐部分をメソッド Z に切り出した．

この例の合計サイクロマティック数は以下の式 4.4 より 8 となる．

$$(\text{合計サイクロマティック数})8 = (main)2 + (\text{メソッド } A)2 + (\text{メソッド } B)2 + (\text{メソッド } Z)2 \quad (4.4)$$

図 4.3，図 4.4 のように，合計サイクロマティック数では同じアルゴリズムで実装したプログラムに対して異なる数値になる．メソッドの分割方法はプログラマによって異な

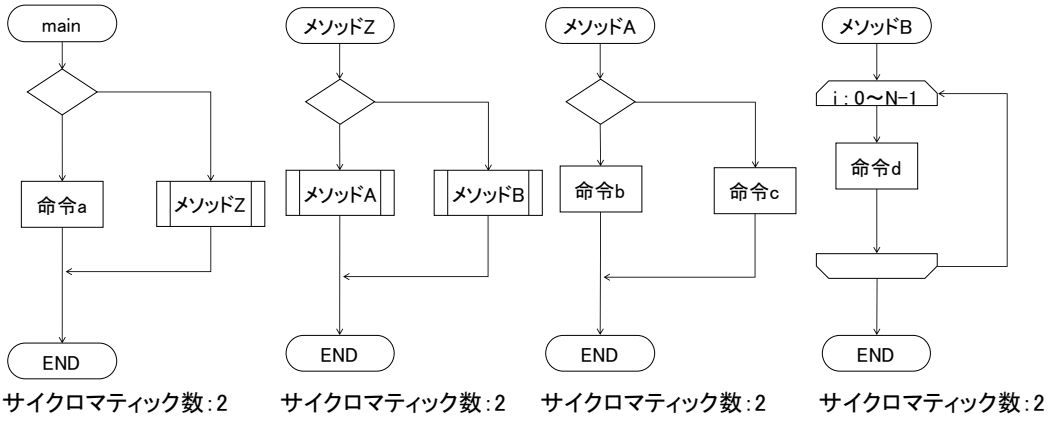


図 4.4: 制御フロー (メソッド 4 つ)

る．しかしメソッドの分割方法によって，同じアルゴリズムでも評価値が異なることはアルゴリズムを評価するメトリクスとして適切ではないと考える．そこで 4.2.3 節ではメソッドの分割方法に依存しないアルゴリズムの複雑度を定量的に表すメトリクスとして拡張サイクロマティック数を提案する．

4.2.3 拡張サイクロマティック数

拡張サイクロマティック数はメソッド呼び出しも含めた全制御フローでのサイクロマティック数とする．拡張サイクロマティック数を求めるには，まず全てのメソッドを 1 つの制御フローに展開する．4.2.2 節で示した，図 4.3，図 4.4 の例を実際に展開した制御フローを図 4.5 に示す．

各メソッドごとに存在する制御フローを展開することで，そのプログラムのアルゴリズムを表す制御フローを得ることができる．展開した制御フローはメソッドの分割方法に依存することなく，アルゴリズムが同じならば，同じ制御フローとなる．拡張サイクロマティック数はこの展開した制御フローに対して，サイクロマティック数を求めたものである．図 4.5 の場合であれば，このアルゴリズムの拡張サイクロマティック数は 5 となる．

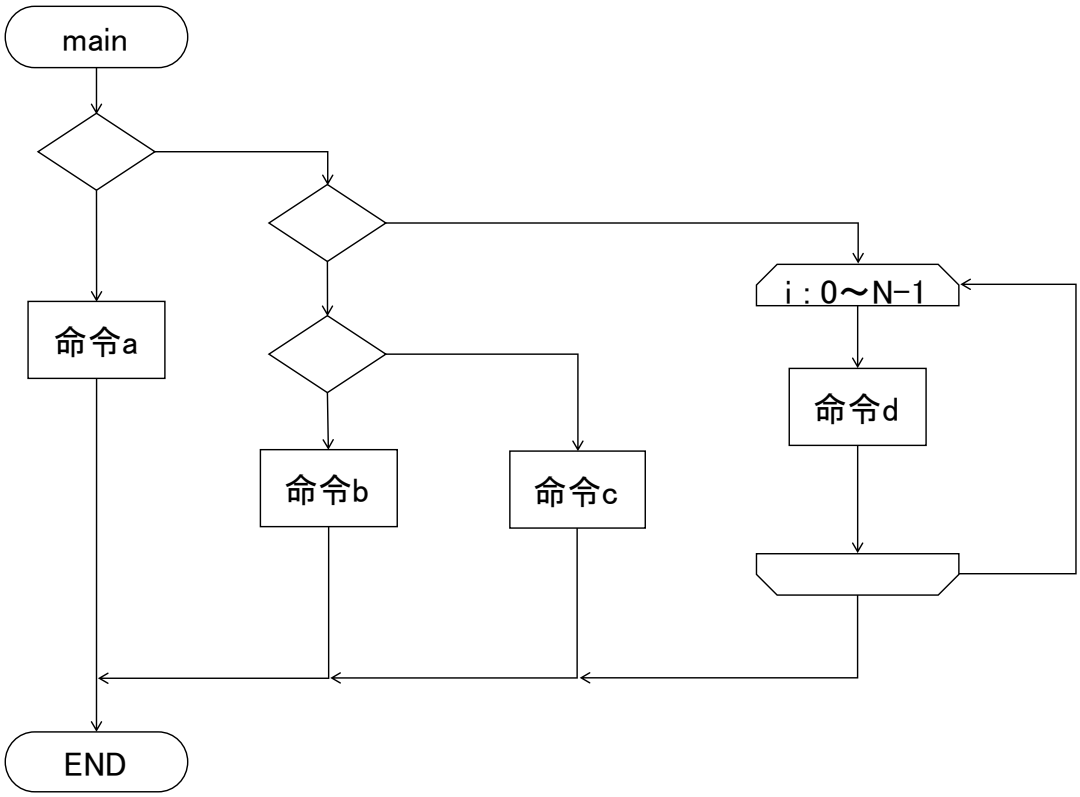


図 4.5: 展開制御フロー

4.3 評価実験

4.2.3 節で提案した拡張サイクロマティック数と可読性の相関を明らかにする．被験者は本学の情報工学科在籍の 5 年から専攻科 2 年の計 10 名である．被験者は 3 名，もしくは 4 名に対して同時に行う．

実験では，同じ仕様で同じ動作をするが，異なるアルゴリズム（異なる拡張サイクロマティック数）で実装された 4 つソースコードを被験者に読んでもらう．プログラムは Java で実装されており，動作は「2 つの数字列を入力し，2 つ目に入力された数字列が 1 つ目に入力された数字列を昇順に並び替えたものであれば「OK」，それ以外なら，「WRONG_ANSWER」と表示する」ものである．

ソースコードは紙に印刷し，4 つのソースコードを 10 分おきに被験者に提示する．被

表 4.1: 実験ソースコードの拡張サイクロマティック数と合計サイクロマティック数

No.	ex	sum
100 点基準	6	6
0 点基準	23	23
1	6	10
2	7	11
3	9	11
4	11	11

験者は 10 分の間にそれぞれのソースコードを読み，可読性を 0〜100 点で採点する．ただし被験者の採点基準を統一するため，4 つのソースコードを読む前に 2 つのソースコードを読んでもらい 1 つを 100 点，1 つを 0 点の基準とする．基準となる 2 つのソースコードにもそれぞれ 10 分間，ソースコードを読む時間を設ける．4 つのソースコードへの採点終了後，被験者に結果集計用紙を配布し，結果を記入してもらう．また，同時にアンケート用紙を配布し，アンケートにも回答してもらう．

基準となる 2 つを含んだ，6 つのソースコードの拡張サイクロマティック数 (ex) と合計サイクロマティック数 (sum) を表 4.1 に示す．100 点の基準には拡張サイクロマティック数，合計サイクロマティック数ともに最小のものとし，0 点の基準にはともに最大のものとした．No.1-4 において拡張サイクロマティック数は 6 から 11 で変化している．拡張サイクロマティック数が大きくなればなるほど，アルゴリズムは複雑であり，可読性は低くなると考えられ，拡張サイクロマティック数と被験者の評価値には負の相関が出ると考えられる．

またソースコード No.1-4 は，合計サイクロマティック数がほぼ同じになるようメソッド分割を行った．このとき，分割の様子が不自然にならないように配慮した．合計サイクロマティック数が同じであっても，可読性には変化がある場合，合計サイクロマティック数ではアルゴリズムによる可読性を表しきれないこととなる．

表 4.2: レベル別相関係数

Lv.	相関係数
下級者	-0.33
中級者	-0.48
上級者	-0.85

4.4 結果と考察

ソースコードの拡張サイクロマティック数と被験者による可読性の評価値との相関係数は-0.55，合計サイクロマティック数と被験者による可読性の評価値との相関係数は-0.17であった．拡張サイクロマティック数と被験者による可読性の評価値には負の相関が認められる．アルゴリズムによる可読性を表すメトリクスとして，拡張サイクロマティック数は有用であると考えられる．

一方で，合計サイクロマティック数と被験者による評価値には相関がほとんどない．これは，合計サイクロマティック数が同じでも，可読性は変化するということであり，同時に合計サイクロマティック数ではアルゴリズムによる可読性を表しきれないことがわかる．

3.4.1 節と同様にアンケートにより被験者のプログラミングレベルを下級者，中級者，上級者に分けてレベルごとに分析を行った．レベル別，拡張サイクロマティック数と被験者による可読性の評価値との相関係数を表 4.2 に示す．

上級者は拡張サイクロマティック数と可読性の評価値に-0.85 と強い負の相関があることがわかる．上級者が感じる可読性を表すメトリクスとして拡張サイクロマティック数は有用であるということが考えられる．

5. 終わりに

本研究では，ソースコードの可読性を定量的に評価する方法を提案した．視覚的可読性では，ソースコードのインデント情報を，ネスト構造 (3 種類) とインデントの状態 (3 種類) という観点から 9 種類に分類し，その 9 種類がそれぞれソースコードの可読性に与える重みをコンジョイント分析で求めることで可読性の定量化を行った．また，論理構造的可読性では，アルゴリズムによる可読性の定量化を行った．1 メソッドの複雑さを表すサイクロマティック数を拡張することで，ソースコード全体の複雑度を表すメトリクスを定義し，被験者実験によりその有用性を示した．

これらのメトリクスを用いて，プログラマのソースコードの可読性を定量的に評価できるようになり，教育現場などで，可読性についての評価を定量的に学生にフィードバック行うことが可能となる．フィードバックを受けた学習者は可読性を意識したコーディングを習得することができると考えている．

今後の課題としては，変数名やコメントなど他の要素についての可読性を定量化することがあげられる．また，本研究での被験者が学生に絞られていることから，被験者をより開発現場に近いプログラマで行うことで有用性を示すことができると考えている．

謝辞

本研究を進めるにあたり，多くの方々のご助力をいただきました．この場を借りて，お礼を申し上げます．指導教員である上野先生には，研究にあたって助言をいただくばかりでなく，進路やなどの面でも助けて下さいました．ありがとうございました．

また，被験者実験に協力いただいた学生の方々に感謝いたします．ありがとうございました．

参考文献

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

[2] 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 Mainenance Tasks, IEEE Trans. Software. Eng., Vol.32, No.12, pp.971 - 987 (2006).

[3] 松本 健一, 井上 克郎, 菊野 亨, 鳥居 宏次, “エラー寿命に基づくプログラマ性能の実験的評価－大学環境におけるプログラム開発－,” 電子情報通信学会論文誌D Vol.J71-D No.10，pp1959-1965(1988)

[4] 高田 義広，鳥居 宏次，“プログラマのデバッグ能力をキーストロークから測定する方法，” 電子情報通信学会論文誌 D-I Vol. J77-D-I No9. pp.646-655(1994)

[5] T. McCabe: “A complexity measure,” IEEE Trans. Softw.Eng, SE-2, 4, pp. 308320 (1976).

[6] 菅 民郎（2004）『Excel で学ぶ多変量解析入門』 オーム社.

[7] 鷹治 沙織, 上野 秀剛,“コードレビュー時の読み方教示によるレビュー効率の変化,” 電子情報通信学会技術研究報告. KBSE, 知能ソフトウェア工学 114(128), 1-8, 2014-07-02