卒　業　研　究　報　告　書

令和元年度

研究題目

プログラミング能力が高い開発者の
コーディングスタイルの特徴の調査

指導教員　　上　野　秀　剛　　准教授

氏　　名　　三　野　天　羽

令和2年1月30日　提出

奈 良 工 業 高 等 専 門 学 校　　情 報 工 学 科

プログラミング能力が高い開発者のコーディングスタイルの特徴の調査

上野研究室　三野天羽

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いわれる．さらに，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．そのため，可読性向上によりソースコードの理解にかかる時間を短縮することは，保守作業量の削減につながると考えられ重要である．しかし，大学等のプログラミング教育において，可読性はプログラムが正しく動作するかに比べて重要視されないことが多い．原因として，可読性が低くても正しく動作するプログラムを作成することは可能なことや，可読性を定量的に評価をすることが難しいからだと考える．そこで本研究では，初級者が書くソースコードの可読性を評価し上級者との違いを示すことで，初級者が可読性を意識したコーディングを習得できると考えた．そのために，コーディングスタイルを定量的に計測するメトリクスを算出し，その値が初級者と上級者でどのように異なるのかを分析した．分析では世界最大規模の競技プログラミングコンテストサイトであるAtCoderに提出されたソースコードを用いた．メトリクスごとにRyanの方法による多重比較を行い，どのグループ間で平均値に有意な差があるのかを確認し，有意差のあったグループの傾向でメトリクスを分類し分析した．その結果，上級者は他の開発者に比べて1行あたりの識別子の数が少ないことがわかった．上級者は，処理を複数行に分け1行の処理を簡潔にするコーディングスタイルであるためと考えられる．また，上級者は他の開発者に比べて1行あたりのカッコの平均の数や最大の数などが多いことがわかった．上級者が作成するソースコードには，改行がなく1行で行う処理が多いテンプレート部が存在することが多い傾向にあるためと考えられる．初級者は他の開発者に比べて1行あたりのインデントの数が多いことがわかった．初級者が作成するプログラムは，他の開発者が作成するプログラムに比べアルゴリズムが複雑である傾向があるためと考えられる．

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いわれる[1]. さらに，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[2]. そのため，可読性向上によりソースコードの理解にかかる時間を短縮することは，保守作業量の削減につながると考えられ重要である.

しかし，大学等のプログラミング教育において，可読性はプログラムが正しく動作するかに比べて重要視されないことが多い. 原因として，可読性が低くても正しく動作するプログラムを作成することは可能なことや，可読性を定量的に評価をすることが難しいからだと考える. 可読性に影響を与える視覚的な要因には，インデントや空行の入れ方などがあり，ソースコードの意味的なまとまりをわかりやすく表現する目的で使用される. このようなソースコードを読みやすくするために，見た目をどのようにするか定めたものをコーディングスタイルと呼ぶ.

本研究では，プログラミング能力の高い開発者が書くソースコードのコーディングスタイルにどのような特徴があるのかを明らかにする. そのために，コーディングスタイルを定量的に計測するメトリクスを算出し，その値が初級者と上級者でどのように異なるのかを分析する. 上級者が書くソースコードのコーディングスタイルの特徴を表すメトリクスを用いることで，ソースコードの可読性を定量的に評価することができる. 初級者が書くソースコードの可読性を評価し，上級者との違いを示すことで，可読性を意識したコーディングを習得できると考えられる.

分析では世界最大規模の競技プログラミングコンテストサイトであるAtCoderに提出されたソースコードを用いる. ソースコードのコーディングスタイルは，1行あたりのインデント·変数の数の平均値などを用いて数値化する. また，開発者のプログラミング能力は，ソースコードを提出したコンテストの成績に基づいたレーティングを用いて判断する.

本研究で得られる上級者が書くソースコードのコーディングスタイルの特徴を用いることで，開発者の能力判定をプログラムの出力が正しいかだけでなく，ソースコードの可読性が高いかという観点でも行うことができる. 可読性を大学などで利用が進みつつある自動採点システムで評価することが可能になれば，可読性を重要視したプログラミング教育を行うことができるため，教育の質の向上に有用だと考えられる.

2 関連研究

2.1 ソースコードの可読性評価

佐々木ら[3]は，変数の定義と参照の間の距離に着目し，ソースコードの可読性向上を目的としたプログラム文の並び替え手法を提案している．提案手法をJavaで記述された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可読性の評価実験を行ったところ，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．この研究では，変数の定義と参照の間の距離と可読性の関係について調査している．本研究では，コーディングスタイルを表す様々なメトリクスと可読性の関係について調査する．

Buseら[4]は，Javaのソースコードの断片100個に対して，学生120人による可読性の評価実験を行っている．得られた可読性の点数と，ソースコード中の変数の長さや空行の数の平均値･最大値などの特徴量との相関を求めている．この研究では多くの特徴量を1つのメトリクスで評価している．対して本研究では，特徴それぞれに対する可読性をフィードバックすることで，学習に役立てられるようにする．

2.2 ソースコード品質の定量的評価

鈴木ら[5]は，メソッドに付けられた名前に着目し，バグ修正の多さとどのような関係があるか調査をしている．5つの著名なオープンソースソフトウェアを対象とし分析を行ったところ，長過ぎる名前やlowerCamelCaseに従っていない名前を持ったメソッドの方がバグ修正が起こりやすいといった傾向が確認された．

阿萬[6]は，ソースコード中のコメントに着目し，フォールトの潜在率とどのような関係があるか調査をしている．7種類のオープンソースソフトウェアを対象とし分析を行ったところ，コメント記述の多いコードほどフォールト潜在率も高いという傾向が確認された．また，コメントアウトもあわせて登場する場合はその可能性がさらに高いという結果も得られた．

これらの研究はいずれもソースコードの特徴と品質の関係について調査している．本研究でもソースコードの特徴に着目し，可読性との関係について調査する．

3 準備

3.1 可読性とコーディングスタイル

可読性とは，開発者がソースコードを読んだときに，そのソースコードの処理の内容や目的をどれだけ理解しやすいかを表す性質である．可読性を左右する要因には論理的構造によるものと，視覚的なものがある．

論理的構造はそのソースコードがどのようなアルゴリズムで実装されているかということで，難解な実装方法であると可読性は低下する．3つの値の最大値を求めるプログラムでアルゴリズムが異なる例をソースコード1とソースコード2に示す.

ソースコード1 最大値1

```
1 int[] ns = {1, 2, 3};
2 int max = ns[0];
3
4 for (int i = 0; i < 3; i++) {
5   if (max < ns[i]) {
6     max = ns[i];
7   }
8 }
```

ソースコード2 最大値2

```
1  int a = 1;
2  int b = 2;
3  int c = 3;
4  int max;
5
6  if (a < b) {
7    if (c < b) {
8      max = b;
9    }
10   else {
11     max = c;
12   }
13 }
14 else {
15   if (a < c) {
16     max = c;
17   }
18   else {
19     max = a;
20   }
21 }
```

ソースコード1は3つの値を配列に格納し，変数maxと順に比較するというアルゴリズムである．ソースコード2は3つの値をそれぞれ変数a, b, cとして格納し，大小関係をif文を用いて場合分けし変数maxに格納するというアルゴリズムである．ソースコード1は処理が簡潔に書かれており可読性が高いのに対し，ソースコード2は条件分岐が複雑で可読性が低い．

視覚的なものには，1行あたりの識別子の数などがある．例として，同じ動作をするが1行あたりの識別子の数が異なるプログラムを示す. ソースコード3に数が少ない例を，ソースコード4に数が多い例を示す.

ソースコード3 条件判定1

```
1 int a = 2;
2 int b = 3;
3 int c = 4;
4
5 int na = a * c;
6 int nb = b * c;
7
8 int ans = na * nb - na;
```

ソースコード4 条件判定2

```
1 int a = 2;
2 int b = 3;
3 int c = 4;
4
5 int ans = a * c * b * c - a * c;
```

ソースコード3は5, 6行目でna, nbに変数をまとめているため8行目に含まれる識別子はans, na, nb, naの4つである．一方，ソースコード4は変数をまとめていないため5行目に含まれる識別子はans, a, c, b, c, a, cの7つである．ソースコード4のansを求める処理はどのような計算を行っているかわかりにくいが，ソースコード3はどのような計算を行っているかがわかりやすいため，可読性が高いといえる．

また，ソースコードを読みやすくするため，見た目をどのようにするか定めたものをコーディングスタイルと呼ぶ．

3.2 AtCoder

AtCoder[1]は世界最大規模の競技プログラミングコンテストサイトで，国内で60,000名，全世界で110,000名が登録している．競技プログラミングとは，与えられた問題に対してどれだけ正確に素早くプログラムを作成できるかを競う競技である．コンテストは上級者向けと初級者・中級者向けで分類されており，制限時間や問題の難易度が異なる．表1にコンテストの分類を示す．

表1 コンテストの分類

分類	コンテスト名	制限時間(h:m)	点数
上級者向け	AGC	2:00〜2:30	300〜2200
初級者・中級者向け	ABC	1:40〜2:00	100〜600

問題の難易度は点数で表され，そのコンテスト内で点数の低い順にA問題，B問題，C問題...と呼ぶ．

ユーザーがコンテスト時間内に提出したソースコードはジャッジシステムにより評価され，表2に示すように分類される．そして，正解した問題の難易度と提出時間の速さにより，そのコンテストのパフォーマンスという値が決定する．パ

[1]AtCoder : https://atcoder.jp/

フォーマンスは，ユーザーのプログラミング能力を表した値であり，大きいほどプログラミング能力が高いことを示す.

表 2 評価の分類

評価	説明
CE (Compilation Error)	提出されたプログラムのコンパイルに失敗しました。
MLE (Memory Limit Exceeded)	問題で指定されたメモリ制限を超えています。
TLE (Time Limit Exceeded)	問題で指定された実行時間以内に プログラムが終了しませんでした。
RE (Runtime Error)	プログラムの実行中にエラーが発生しました。 コンパイル時に検知できなかったエラーがあります. スタックオーバーフロー、ゼロ除算などが原因です.
OLE (Output Limit Exceeded)	問題で指定された制限を超えるサイズの出力を行いました。
IE (Internal Error)	内部のエラー、つまりジャッジシステムのエラーです。
WA (Wrong Answer)	誤答です。提出したプログラムの出力は正しくありません。
AC (Accepted)	正答です。運営が用意したテストを全てパスし、 正しいプログラムであると判定されました。

4 実験

分析に用いるソースコードを収集する．また，コーディングスタイルを表すためどのようなメトリクスを用いるか決定する．

4.1 対象ソースコード

表3に示す2929件のソースコードを対象とする．

表3 対象とするソースコード

コンテスト名	問題	言語	提出時間	評価
ABC140	A問題	Java8 (OpenJDK 1.8.0)	コンテスト時間内	AC以外
AGC1~40	A問題	Java8 (OpenJDK 1.8.0)	コンテスト時間内	すべて

初級者の提出するソースコードを得るために，初級者向けコンテストであるABCの最も難易度の低い問題であるA問題で，ACではなかった提出を収集する．ABCのA問題は簡単な文法の確認問題で，ABC140ではユーザーの約99.3%が正解する問題であるため，この問題を誤答するユーザーは初級者が多いと考えた．中級者･上級者の提出するソースコードを得るために，上級者向けコンテストであるAGCの1~40のA問題の提出を対象とする．AGCのA問題はABCのA問題に比べ難易度が上がり，AGC40ではユーザーの約15.8%が誤答する問題であった．そのため，この問題を誤答するユーザーにも中級者･上級者は存在すると考え，提出されたすべてのソースコードを収集する．

収集したソースコードには同一ユーザーがコンテスト中に複数回提出したものが含まれている．コンテスト中に同一ユーザーのコーディングスタイルが大きく変化することはないと考える．分析の際に提出回数の多いユーザーのコーディングスタイルの影響が大きくなることを防ぐため，ユーザーがある問題に対して最終提出したソースコードを対象とする．

4.2 メトリクス

プログラマの能力を表す指標として，ソースコードを提出したユーザーがそのコンテストで得たパフォーマンスを用いる．ソースコードのコーディングスタイルを表す指標として使用するメトリクスを表4に示す.

表 4 使用するメトリクス

メトリクス名	説明
avg_identifiers	1 行に含まれる識別子の数の平均値
max_identifiers	1 行に含まれる識別子の数の最大値
avg_line_len	1 行の長さの平均値
max_line_len	1 行の長さの最大値
avg_blank_lines	ソースコードに含まれる空行の割合
max_blank_lines	最大でいくつの空行が並んでいるか
avg_bracket	1 行に含まれる括弧 (“(”, “{”) の数の平均値
max_bracket	1 行に含まれる括弧 (“(”, “{”) の数の最大値
avg_period	1 行に含まれるピリオドの数の平均値
max_period	1 行に含まれるピリオドの数の最大値
avg_comma	1 行に含まれるコンマの数の平均値
max_comma	1 行に含まれるコンマの数の最大値
avg_space	1 行に含まれるスペースの数の平均値
max_space	1 行に含まれるスペースの数の最大値
avg_equal	1 行に含まれるイコールの数の平均値
max_equal	1 行に含まれるイコールの数の最大値
avg_keywords	1 行に含まれるキーワードの数の平均値
max_keywords	1 行に含まれるキーワードの数の最大値
avg_indent	1 行に含まれるインデントの数の平均値
max_indent	1 行に含まれるインデントの数の最大値

Buse ら [4] は 1 行に含まれる識別子・記号などの数の平均値や最大値と可読性の関係を調査している．その結果，可読性とメトリクスに相関が見られた．コーディングスタイルは可読性に影響を与えるため，表 4 に示したメトリクスはコーディングスタイルを表す指標として機能すると考える．本研究では分析対象とする個々のソースコードからメトリクスの値を算出し，ソースコードのコーディングスタイルを表す指標とする．

4.3 分析方法

各メトリクスごとに極端に値が大きく，かつその値を計測する数が少ないソースコードは，通常ではないコーディングスタイルであると考え省いた．外れ値を省いたソースコードをパフォーマンスで昇順にソートし，20 個のグループに数が等しくなるように分割する．これにより，グループ 1 は初級者，グループ 20 は上級者というように分類される．分割したグループごとに，各種メトリクスの平均値に差があるかを確認するため一元配置分散分析を行う．一元配置分散分析は三群間以上の平均値の差を検定するための手法である．

5 結果と考察

5.1 一元配置分散分析

表5にメトリクスごとの一元配置分散分析の結果を示す. 一元配置分散分析の結果，avg_blank_lines以外の全てのメトリクスでグループごとの平均値に有意差（p <0.01）があった. 以上の結果から，開発者の能力によりコーディングスタイルに差があるといえる.

表5メトリクスごとの一元配置分散分析結果

メトリクス名	p値
avg_identifiers	0.000**
max_identifiers	0.000**
avg_line_len	0.000**
max_line_len	0.000**
avg_blank_lines	0.340
max_blank_lines	0.009**
avg_bracket	0.000**
max_bracket	0.000**
avg_period	0.000**
max_period	0.000**
avg_comma	0.000**
max_comma	0.000**
avg_space	0.000**
max_space	0.000**
avg_equal	0.000**
max_equal	0.000**
avg_keywords	0.000**
max_keywords	0.000**
avg_indent	0.000**
max_indent	0.000**

* : p <0.05, ** : p <0.01

5.2 メトリクス別分析

一元配置分散分析により，多くのメトリクスでグループごとの平均値に差があることが確認できた. そこで，どのグループ間で平均値に有意な差があるか確認するため，メトリクスごとにRyanの方法による多重比較を行う. 多重比較は3つ以上の群で，どの群間に平均値の差があるかを検定する場合に用いる手法で，Ryanの方法はその1つである. また，有意差のあったグループの傾向でメトリクスを分類し分析する.

以降での多重比較の結果を示すヒートマップの見方を説明する．図1にヒートマップの例を示す.

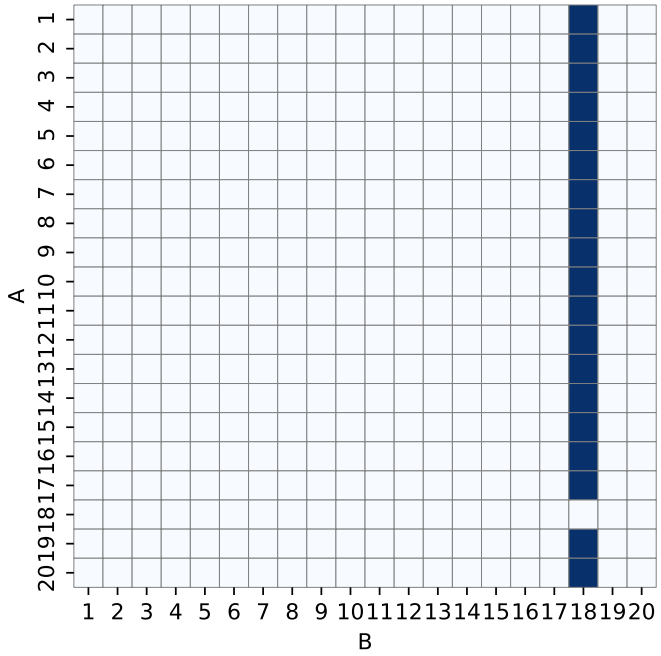

図 1 ヒートマップの例

図の縦軸と横軸はグループの番号を示し，値が高いほど開発者のプログラミング能力が高いことを表す. 縦軸をグループA, 横軸をグループBとすると，色のついているセルはグループAに対してグループBは値が有意に高いことを表す. 例として，図1の縦軸が1, 横軸が18のセルに色がついているのは，グループ18の値がグループ1の値よりも有意に高いことを表す. 図1では横軸が18のセル全てに色がついているので，グループ18は他グループに対して有意に値が高いと解釈できる.

5.2.1 グループ20の値が低いメトリクス

グループ20の値が低い傾向のあったメトリクスの多重比較の結果を示す. 図2, 図3にメトリクスavg_identifiers, avg_equalの多重比較の結果を示す. それぞれのメトリクスで，グループ20に対して他のグループの値が高い，すなわちグループ20の値が低いことがわかる.

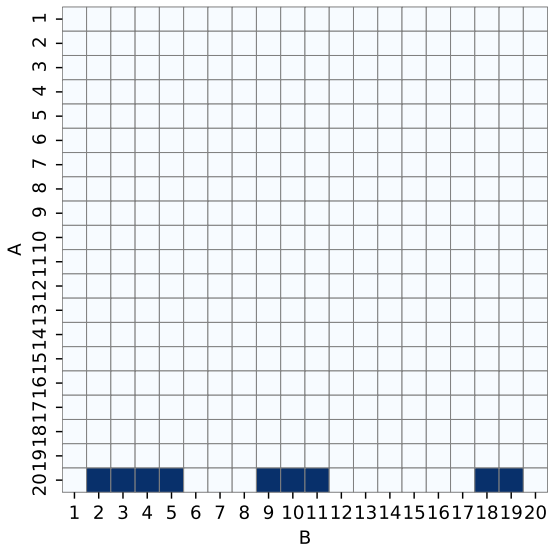

図 2 avg_identifiers の多重比較結果

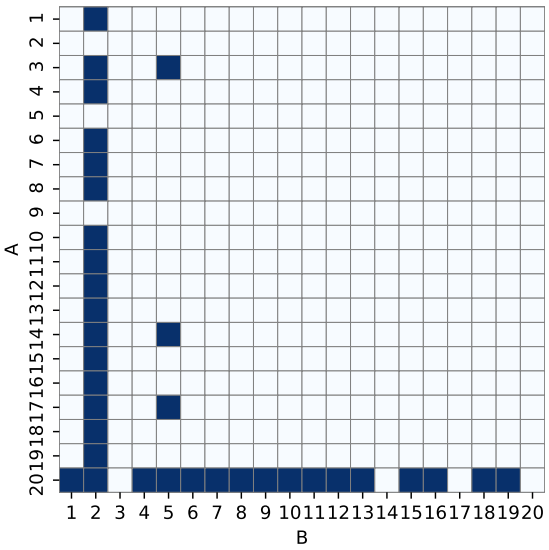

図 3 avg_equal の多重比較結果

図2よりグループ20の上級者は他の開発者に比べて1行あたりの識別子の数が有意に少ないことがわかる．1行あたりの識別子の数がソースコードのコーディングスタイルに与える影響を調べるため，値の低い例と高い例を比較する．

ソースコード5にグループ20でavg_identifiersの値が1.29と低い例を，ソースコード6にグループ20以外でavg_identifiersの値が2.26と高い例の一部を示す．どちらのソースコードも問題「{重さ0.25, コストQ},{重さ0.5, コストH},{重さ1, コストS},{重さ2, コストD}の品物が無限個あるとき，重さをNにするために必要な最小のコストを求めよ．」に対して提出された．

ソースコード6はコストを計算する処理を15行目に書いている．一方，ソースコード5は変数ansを定義し，同じ処理を13〜22行目に分けて書いている．そのため1行あたりに用いる識別子の数がソースコード6の15行目は11に対して，ソースコード5の13〜22行目は1.8と少なくなっている．

グループ20の開発者は，処理を複数行に分け1行の処理を簡潔にするコーディングスタイルであるため，他のグループの開発者に比べて1行あたりの識別子の数が少なくなったと考えられる．

ソースコード5 グループ:20, avg_identifiers:1.29

```
void submit() {
  int q = nextInt();
  int h = nextInt();
  int s = nextInt();
  int d = nextInt();

  long n = nextLong() * 4;

  h = Math.min(h, 2 * q);
  s = Math.min(s, 2 * h);
  d = Math.min(d, 2 * s);

  long ans = 0;  ←変数ans 定 義
  ans += (n / 8) * d;  ←コスト計算処理開始
  n %= 8;

  ans += (n / 4) * s;
  n %= 4;

  ans += (n / 2) * h;
  n %= 4;
  ans += n* q;  ←コスト計算処理終了

  out.println(ans);
}
```

ソースコード6 グループ:19, avg_identifiers:2.26

```
static class A {
  public void solve(int testNumber, Scanner in, PrintWriter out) {
      long q = in.nextLong();
      long h = in.nextLong();
      long s = in.nextLong();
      long d = in.nextLong();
      if (2 * q < h) {
          h = 2 * q;
      }
      if (2 * h < s) {
          s = 2 * h;
      }
      long n = in.nextLong();
      out.println(Math.min(s * n, n / 2 * d + (n % 2) * s));  ←コスト計算処理
  }
}
```

avg_equalに着目すると，図3よりグループ20の上級者は他の開発者に比べて1行あたりのイコールの数が有意に少ないことがわかる．1行あたりのイコールの数がソースコードのコーディングスタイルに与える影響を調べるため，値の低い例と高い例を比較する．

ソースコード7にグループ20でavg_equalが0.18と低い例を，ソースコード8とソースコード9にグループ20以外でavg_equalがどちらも0.83と高い例を示す．いずれもソースコードの一部のみを示す．

ソースコード 7 グループ:20, avg_equal:0.18

```
「メイン処理部」
void solve() throws IOException {
    String s = nextToken();
    int balance = 0;
    int ans = s.length();
    for (char c : s.toCharArray()) {
      if (c == 'S') {
        balance++;
      } else {
        if (balance > 0) {
          ans -= 2;
          balance--;
        }
      }
    }

    out.println(ans);
}

「テンプレート部 (代入処理や条件判定が含まれていない)」
  int nextInt() throws IOException {
    return Integer.parseInt(nextToken());
  }

  long nextLong() throws IOException {
    return Long.parseLong(nextToken());
  }

  double nextDouble() throws IOException {
    return Double.parseDouble(nextToken());
  }
}
```

ソースコード 8 グループ:17, avg_equal:0.83

```
「メイン処理部」
void solve() throws IOException{
  int n = in.nextInt();
  String s = in.nextToken();
  String t = in.nextToken();
  out:for(int i=0; i<=n; i++){
    char[] ss = (s+getSub(t, i)).toCharArray();
    for(int j=0; j<n; j++){
      if(ss[j]!=s.charAt(j)) continue out;
    }
    for(int j=0; j<n; j++){
      if(ss[ss.length-1-j]!=t.charAt(t.length()-1-j)) continue out;
    }
    System.out.println(n+i);
    return;
  }
}

「テンプレート部 (代入処理や条件判定が含まれている)」
static long pow(long a,int n){long r=1;while(n>0){if((n&1)==1)r*=a;a*=a;n>>=1;}return r;}
static String itob(int a,int l){return String.format("%"+l+"s",Integer.toBinaryString(a
    )).replace(' ','0');}
static void sort(int[]a){m_sort(a,0,a.length,new int[a.length]);}
static void sort(int[]a,int l){m_sort(a,0,l,new int[l]);}
static void sort(int[]a,int l,int[]buf){m_sort(a,0,l,buf);}
static void sort(int[]a,int s,int l,int[]buf){m_sort(a,s,l,buf);}
static void m_sort(int[]a,int s,int sz,int[]b)
{if(sz<7){for(int i=s;i<s+sz;i++)for(int j=i;j>s&&a[j-1]>a[j];j--)swap(a, j, j-1);return
    ;}
m_sort(a,s,sz/2,b);m_sort(a,s+sz/2,sz-sz/2,b);int idx=s;int l=s,r=s+sz/2;final int le=s+
    sz/2,re=s+sz;
while(l<le&&r<re){if(a[l]>a[r])b[idx++]=a[r++];else b[idx++]=a[l++];}
while(r<re)b[idx++]=a[r++];while(l<le)b[idx++]=a[l++];for(int i=s;i<s+sz;i++)a[i]=b[i];
```

ソースコード9 グループ:1, avg_equal:0.83

```
「メイン処理部」
public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    String S = sc.next();
    int [] list = new int [S.length()];
    for(int a=0;a<S.length();a++){
        if(S.charAt(a)=='B') list[a]=0;
        else list[a]=1;
    }
    int num=0;
    int check=1;
    int a=0;
    while(check!=0){
        check=0;
        for(int b=a;b<S.length()-1;b++){
            if (list[b] == 0 && list[b + 1] == 1) {
                list[b] = 1;
                list[b + 1] = 0;
                num++;
            }
        }
        for(int b=a;b<S.length()-1;b++) if(list[b]==1)check++;
        if(list[a]==1) a++;

    }
    System.out.println(num);
}

「テンプレート部」
なし
```

ソースコードはメイン処理部とテンプレート部で構成されている．メイン処理部は開発者が問題を解くために必要な処理をコンテスト時間内に記述した部分である．一方，テンプレート部は標準入力や配列のソートなど，問題に関係なく必要な処理の呼び出しを単純化するためコンテスト前に記述した部分である．特に上級者において，問題の種類にかかわらず事前に用意したテンプレートを用いているケースが多数見られた．

3つのソースコードのメイン処理部に大きなコーディングスタイルの違いは見られないが，テンプレート部のコーディングスタイルに違いが見られる．ソースコード7とソースコード8にはいずれもテンプレート部が含まれており，問題とは関係のない処理によってソースコードの行数が増えている．ソースコード7のテンプレートはメソッド呼び出しを単純化するための処理が多く，代入や条件判定をする処理が含まれていないため，メイン処理部を含んだソースコード全体の1行あたりのイコール数は少ない．

一方，ソースコード8のテンプレートには代入処理や条件判定が含まれており，ソースコード全体の1行あたりのイコール数は変化しない．また，ソースコード9にはテンプレート部が存在しないため，全体の行数が増えず，平均値が低くならなかった．よって，メトリクス avg_equal はテンプレート部のコーディングスタイルの影響を受けていると考えられる．

ソースコードの性質を正確に理解するためには，問題の処理に必要な部分のみを評価することが望ましい．そのためには，ソースコードのメイン処理部とテンプレート部を区別する手法を用いてメイン処理部のみを評価する必要がある．

5.2.2 グループ2, 5の値が高いメトリクス

グループ2, 5の値が高い傾向のあったメトリクスの多重比較の結果を示し考察する.

図4にメトリクスavg_indentの多重比較の結果を示す.

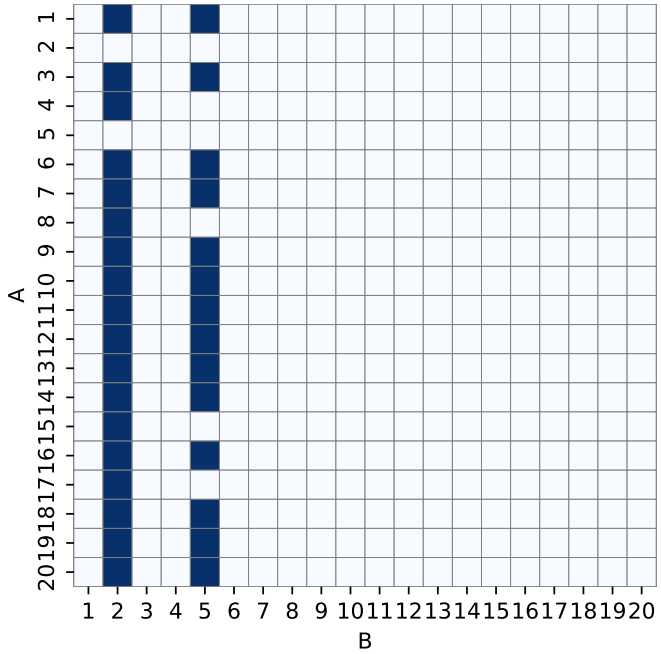

図 4 avg_indentの多重比較結果

グループ2, 5の初級者は他の開発者に比べて1行あたりのインデントの数が多いことがわかる. 1行あたりのインデントの数がソースコードのコーディングスタイルに与える影響を調べるため, 値の低い例と高い例を比較する.

ソースコード10にグループ2でavg_indentの値が8.08と高い例を, ソースコード11にグループ20以外でavg_indentの値が4.14と低い例の一部を示す.

ソースコード 10 グループ:2, avg_indent:8.08

```
1  public static void main(String args[]) throws FileNotFoundException{
2    Scanner sc=new Scanner(System.in);
3    int n=Integer.parseInt(sc.nextLine());
4    String[] buf=sc.nextLine().split(" ");
5    int[] num=new int[buf.length];
6
7    for(int i=0; i<buf.length; i++){
8      num[i]=Integer.parseInt(buf[i]);
9    }
10
11   int idx=1;
12   int cnt=1;
13   int ans=0;
14
15   while(idx<num.length){  ←ループ処理 1開始
16     cnt=1;
17     if(num[idx-1]!=num[idx]){
18       idx++;
19     }else{
20       while(idx<num.length && num[idx-1]==num[idx]){  ←ループ処理 2開始
21         idx++;
22         cnt++;
23       }  ←ループ処理 2終了
24     }
25     if(cnt>1){
26       ans+=cnt/2;
27     }
28   }
29   System.out.println(ans);
30 }  ←ループ処理 1終了
```

ソースコード 11 グループ:20, avg_indent:4.14

```
1  public void solve(int testNumber, InputReader in, OutputWriter out) {
2    int n = in.nextInt();
3    int[] a = new int[n];
4    for (int i = 0; i < n; ++i) {
5      a[i] = in.nextInt();
6    }
7    int res = 0;
8    for (int i = 1; i < n; ++i) {  ←ループ処理開始
9      if (a[i] == a[i - 1] && a[i - 1] != -1) {
10       a[i] = -1;
11       ++res;
12     }
13   }
14   out.printLine(res);
15 }  ←ループ処理終了
```

どちらのソースコードも問題「1〜10000の整数からなる数列が与えられたとき，数列の任意の箇所を1〜10000の数字に置き換え，数列のどの隣り合う数も異なる状態にする．置き換えるべき最小の回数を答えよ．」に対して提出された．

ソースコード11は8〜13行目にあるループ処理の内側に他のループ処理が存在しないため，計算量が線形時間のアルゴリズムである．一方，ソースコード10は13〜31行目のループ処理1の内側に20〜23行目のループ処理2が存在するので，計算量が二乗時間のアルゴリズムである．どちらも正しい出力をするが，グループ2の開発者が作成したプログラムはグループ20の開発者が作成したプログラムに対してアルゴリズムが複雑であると言える．そのため1行あたりのインデント数がソースコード11は4.14に対して，ソースコード10は8.08と高くなっている．

ソースコード12にグループ2でavg_indentの値が10.92と高い例を，ソースコード13にグループ20以外でavg_indentの値が4.25と低い例の一部を示す．

ソースコード 12 グループ:5, avg_indent:10.92

```
1  public static void main(String[] args) {
2    Scanner sc = new Scanner(System.in);
3    int N = sc.nextInt();
4    int A[] = new int[N];
5    for (int i = 0; i < N; i++) {
6      A[i] = sc.nextInt();
7    }
8    int temp = 0;
9    int count = 0;
10   int ans = 0;
11   int memo[] = new int[N + 1];
12   int n = 0;
13   for (int i = 0; i < N; i++) {  ←ループ処理 1開始
14     for (int j = i; j < N; j++) {  ←ループ処理 2開始
15       if (memo[j] != 0) {
16         count = memo[j];
17         break;
18       } else {
19         temp += A[j];
20         if (temp == 0) {
21           count++;
22           n = j - i - 1;
23         }
24       }
25     }  ←ループ処理 2終了
26     ans += count;
27     memo[i + n] = count - 1;
28     count = 0;
29     n = 0;
30     temp = 0;
31   }  ←ループ処理 1終了
32   System.out.println(ans);
33 }
```

ソースコード 13 グループ:20, avg_indent:4.25

```
1  public void solve(int testNumber, InputReader in, OutputWriter out) {
2    int n = in.nextInt();
3    HashMap<Long, Long> was = new HashMap<>();
4    long cur = 0;
5    was.put(cur, 1L);
6    long res = 0;
7    for (int i = 0; i < n; ++i) {  ←ループ処理開始
8      int x = in.nextInt();
9      cur += x;
10     if (was.containsKey(cur)) {
11       res += was.get(cur);
12     }
13     if (was.containsKey(cur)) {
14       was.put(cur, was.get(cur) + 1);
15     } else {
16       was.put(cur, 1L);
17     }
18   }  ←ループ処理終了
19   out.printLine(res);
20 }
```

どちらのソースコードも問題「-10^9〜10^9の整数からなる数列が与えられたとき，数列の連続する区間でその区間内の値の総和が0になるものが何個あるか答えよ.」に対して提出された．

ソースコード13は7〜18行目にあるループ処理の内側に他のループ処理が存在しないため，計算量が線形時間のアルゴリズムである．一方，ソースコード12は15〜28行目のループ処理1の内側に14〜25行目のループ処理2が存在するので，計算量が二乗時間のアルゴリズムである．どちらも正しい出力をするが，グループ5の開発者が作成したプログラムはグループ20の開発者が作成したプログラムに対してアルゴリズムが複雑であると言える．そのため1行あたりのインデント数がソースコード13は4.25に対して，ソースコード12は10.02と高くなっている．

これらの結果から，グループ2, 5の開発者が作成するプログラムは他グループの開発者が作成するプログラムに比べアルゴリズムが複雑である傾向があるため，1行あたりのインデント数が多くなったと考えられる．

5.2.3 グループ18,19の値が高いメトリクス

グループ20の値が低い傾向のあったメトリクスの多重比較の結果を示し考察する．

図5, 図6, 図7, 図8にメトリクスavg_bracket, avg_line_len, max_comma, max_bracketの多重比較の結果を示す.

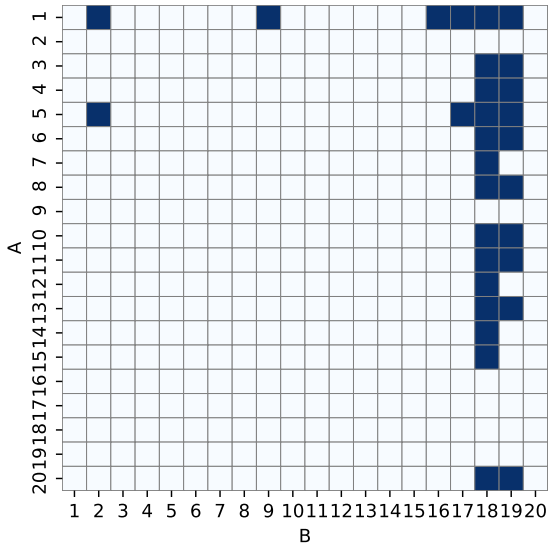

図5 avg_bracketの多重比較結果

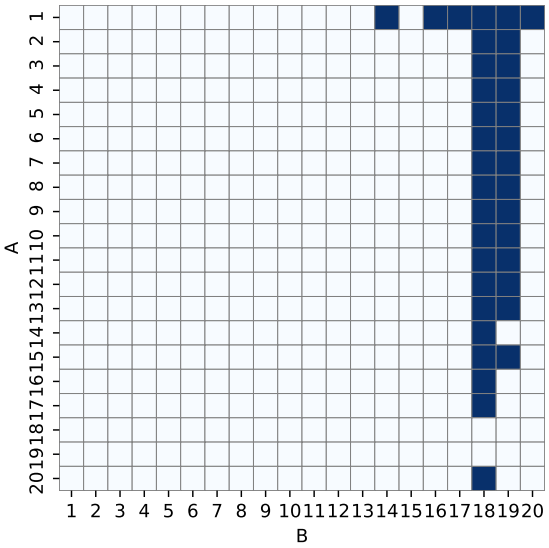

図6 avg_line_lenの多重比較結果

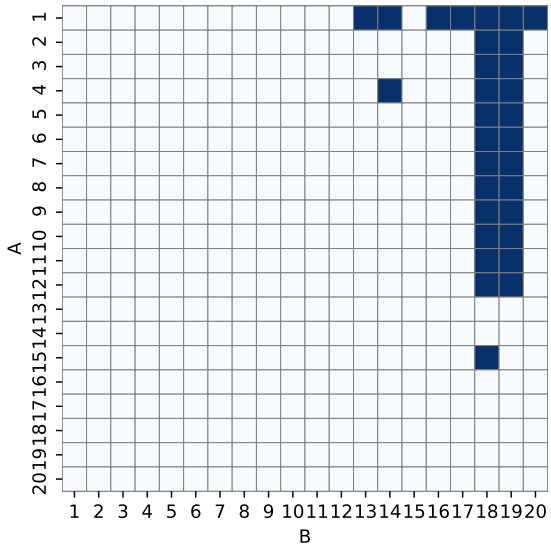

図7 max_commaの多重比較結果

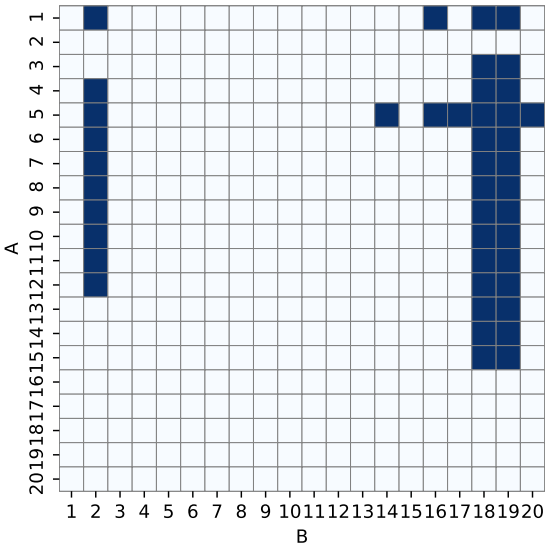

図8 max_bracketの多重比較結果

他グループに対して，グループ18, 19は1行あたりのカッコの数，1行あたりの平均長，1行に含まれるコンマ・カッコの最大数が多いことがわかる．これらのメトリクスがソースコードのコーディングスタイルに与える影響を調べるため，値の低い例と高い例を比較する．

ソースコード14にグループ18でavg_bracket, avg_line_len, max_comma, max_bracketがそれぞれ2.00, 44.18, 8, 8と高い例を，ソースコード15とソースコード16にグループ20以外でメトリクスそれぞれ{(0.56, 16.37, 2, 3), (0.46, 11.14, 1, 2)}と低い例の一部を示す．

ソースコード14 グループ:18, avg_bracket:2.00,avg_line_len:44.18,max_comma:8,max_bracket:8

```
「メイン処理部」
void solve() throws IOException{
  int n = in.nextInt();
  long[] a = new long[n];
  long[] b = new long[n];
  for(int i=0; i<n; i++){
    a[i] = in.nextLong();
    b[i] = in.nextLong();
  }
  long base = 0;
  long ans = 0;
  for(int i=n-1; i>=0; i--){
    a[i] += base;
    if(a[i]%b[i]==0) continue;
    long time = (a[i]/b[i])+1;
    long cnt = time*b[i]-a[i];
    base += cnt;
    ans += cnt;
  }
  System.out.println(ans);
}

「テンプレート部 (処理を 1行で記述している)」
static void dump(int[]a){StringBuilder s=new StringBuilder();for(int i=0;i<a.length;i
    ++)
s.append(a[i]).append(" ");out.println(s.toString().trim());}
static void dump(int[]a,int n){for(int i=0;i<a.length;i++)out.printf("%"+n+"d ",a[i]);
    out.println();}
static void dump(long[]a){StringBuilder s=new StringBuilder();for(int i=0;i<a.length;i
    ++)
s.append(a[i]).append(" ");out.println(s.toString().trim());}
static void dump(char[]a){for(int i=0;i<a.length;i++)out.print(a[i]);out.println();}
static long pow(long a,int n){long r=1;while(n>0){if((n&1)==1)r*=a;a*=a;n>>=1;}return r;}
static String itob(int a,int l){return String.format("%"+l+"s",Integer.toBinaryString(a
    )).replace(' ','0');}
static void sort(int[]a){m_sort(a,0,a.length,new int[a.length]);}
static void sort(int[]a,int l){m_sort(a,0,l,new int[l]);}
```

ソースコード15 グループ:20, avg_bracket:0.56,avg_line_len:16.37,max_comma:2,max_bracket:3

```
「メイン処理部」
static class TaskA {

  public void solve(int testNumber, InputReader in, OutputWriter out) {
    int n = in.nextInt();
    int cnt0 = 0, cnt1 = 0;
    for (int i = 0; i < n; ++i) {
      int x = in.nextInt() % 2;
      if (x == 0) {
        ++cnt0;
      } else {
        ++cnt1;
      }
    }
    out.printLine(cnt1 % 2 == 0 ? "YES" : "NO");
  }

}

「テンプレート部 (改行が含まれる)」
public int nextInt() {
  int sgn = 1;
  int c = readSkipSpace();
  if (c == '-') {
    sgn = -1;
    c = read();
  }
  int res = 0;
  do {
    if (c < '0' || c > '9') {
      throw new InputMismatchException();
    }
    res = res * 10 + c - '0';
    c = read();
  } while (!isSpace(c));
  res *= sgn;
  return res;
}
```

ソースコード16 グループ:10, avg_bracket:0.46,avg_line_len:11.14,max_comma:1,max_bracket:2

```
「メイン処理部」
public class Main
{
  public static void main(String[] args) throws IOException
  {
    BufferedReader r = new BufferedReader(new InputStreamReader(System.in), 1);

    String s;
    String sl[];

    int a[] = new int[300000];

    s = r.readLine();
    int n = s.length();
    for(int i = 0; i < n; i++)
    {
      a[i] = (s.charAt(i) == 'B' ? 1 : 0);
    }

「テンプレート部」
なし
```

3つのソースコードの間でメイン処理部に大きなコーディングスタイルの違いは見られないが，テンプレート部のコーディングスタイルに違いが見られる．

ソースコード14には，改行がなく処理を1行で記述したテンプレート部が存在するため，ソースコード全体の1行あたりのカッコの数，1行あたりの平均長，1行に含まれるコンマ・カッコの最大数が多くなっている．しかし，ソースコード15のテンプレートには改行がありテンプレート部の行数が増えているため，ソースコード全体の平均値や最大値に影響を与えていない．また，ソースコード16に

はテンプレート部が存在しないため，ソースコード全体の平均値や最大値に影響を与えなかった．

よって，メトリクス avg_bracket, avg_line_len, max_comma, max_bracket はテンプレート部のコーディングスタイルの影響を受けており，グループ 18, 19 の開発者が作成するソースコードには，改行がなく処理を 1 行で記述したテンプレート部が存在することが多い傾向にあるため，各メトリクスの値が高くなったと考えられる．

5.2.4 グループ 18,19,20 の値が高いメトリクス

グループ 18, 19, 20 の値が低い傾向のあったメトリクスの多重比較の結果を示し考察する．

図 9, 図 10, 図 11, 図 12, 図 13 にメトリクス max_equal, max_space, max_identifiers, max_line_len, max_period の多重比較の結果を示す.

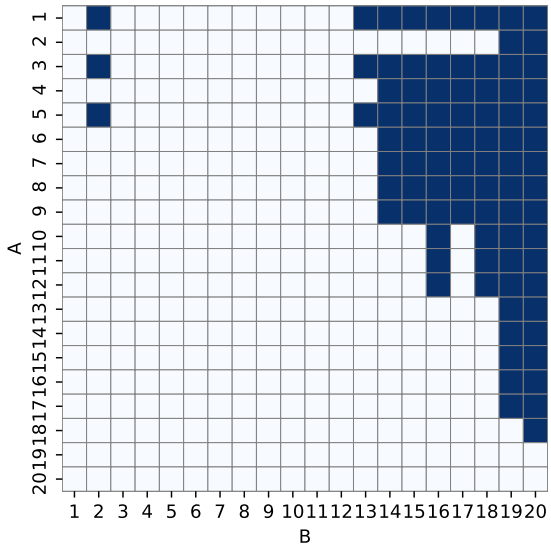

図 9 max_equal の多重比較結果

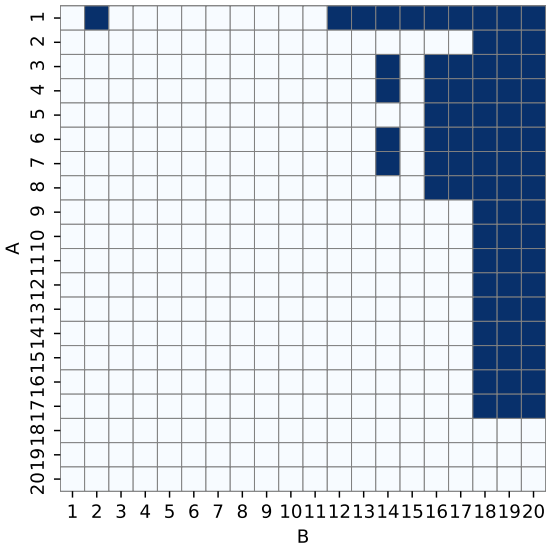

図 10 max_space の多重比較結果

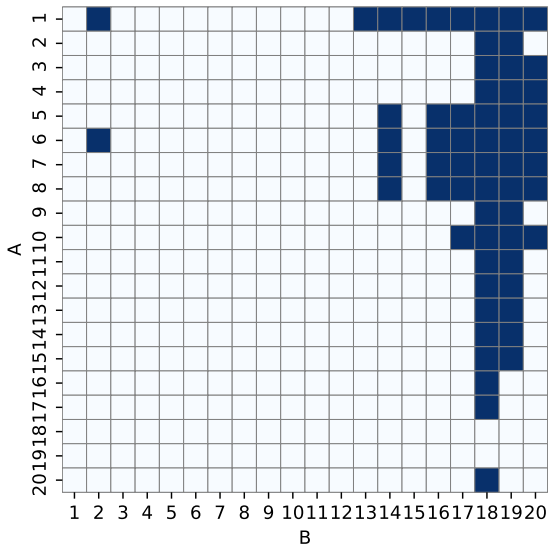

図 11 max_identifiers の多重比較結果

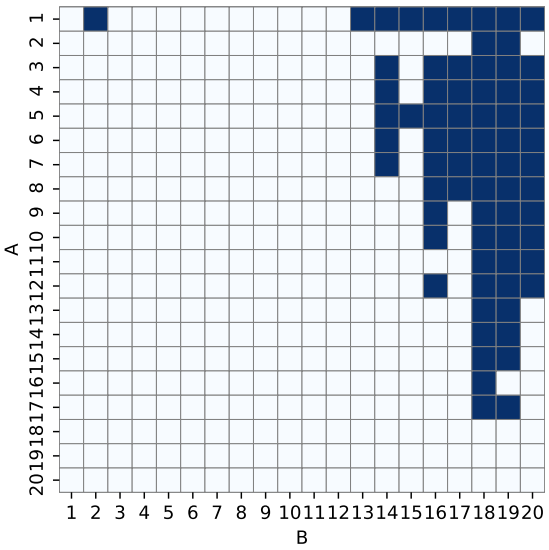

図 12 max_line_len の多重比較結果

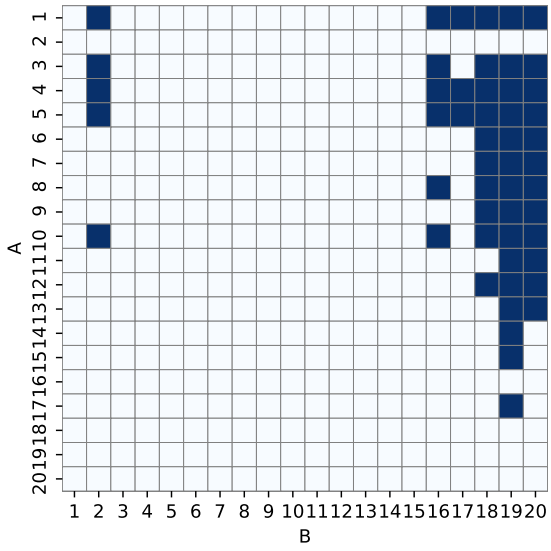

図 13 max_period の多重比較結果

それぞれのメトリクスで，グループ18以上の値が18未満のグループの値に対して高いことがわかる．これらのメトリクスがソースコードのコーディングスタイルに与える影響を調べるため，値の低い例と高い例を比較する．

ソースコード17にグループ18, 19以外でmax_equal, max_space, max_identifiers, max_line_len, max_periodの値が2,9,7,70,2と低い例を，ソースコード18とソースコード19にグループ20, 18でメトリクスの値がそれぞれ{(10,20,12,104,4), (12,57,28,331,6)}と高い例の一部を示す．

ソースコード 17 グループ:8,max_equal:2,max_space:9,max_identifiers:7,max_line_len:70,max_period:2

```
「メイン処理部」
private void calc() throws Exception{
  int N = sc.nextInt();
  int N2 = N * 2 ;
  int[] ary = new int[N2];
  for (int i = 0; i < N2; i++) {
    ary[i] = sc.nextInt();
  }
  Arrays.sort(ary);
  int result = 0;
  for(int j = 0; j < N2; j+=2){
    int num = Math.min(ary[j], ary[j+1]);
    result += num;
  }
  System.out.println(result);
}

「テンプレート部」
public String next() throws IOException {
  if (i < s.length)
    return s[i++];
  String st = br.readLine();
  while (st == "")
    st = br.readLine();
  s = st.split(regex, 0);
  i = 0;
  return s[i++];
}

public int nextInt() throws NumberFormatException, IOException {
  return Integer.parseInt(next());
}
```

ソースコード 18 グループ:20,max_equal:10,max_space:20,max_identifiers:12,max_line_len:104,max_period:4

```
「メイン処理部」
public void solve(int testNumber, InputReader in, OutputWriter out) {
  int n = in.readInt();
  int c = in.readInt();
  int k = in.readInt();
  int[] t = IOUtils.readIntArray(in, n);
  ArrayUtils.sort(t);
  int start = 0;
  int answer = 1;
  for (int i = 0; i < n; i++) {
      if (i >= start + c || t[i] > t[start] + k) {
          answer++;
          start = i;
      }
  }
  out.printLine(answer);
}

「テンプレート部 (改行が含まれる)」
public static boolean isWhitespace(int c) {  ←空白文字の判定を行うメソッド
   return c == ' ' || c == '\n' || c == '\r' || c == '\t' || c == -1;
}
public static void sort(IntList list, IntComparator comparator) {
  quickSort(list, 0, list.size() - 1, (Integer.bitCount(Integer.highestOneBit(list.size
      ()) - 1) * 5) >> 1,  ← 1行で書かざるを得ない処理
        comparator);
}
```

ソースコード19グループ:18,max_equal:12,max_space:57,max_identifiers:28,max_line_len:331,max_period:6

```
「メイン処理部」
public void run() throws IOException {
  //  int TEST_CASE = Integer.parseInt(new String(io.nextLine()).trim());
      int TEST_CASE = 1;
      while(TEST_CASE-- != 0) {
        int n = io.nextInt();
        long a = io.nextInt();
        long b = io.nextInt();
        long ans = 0;
        ans += b * (n - 1) + a;
        ans -= a * (n - 1) + b;
        io.out.println(Math.max(0, ans + 1));
      }
}

「テンプレート部 (処理を 1行で記述している)」
static { for(int i = 0; i < 10; i++) { isDigit['0' + i] = true; } isDigit['-'] = true;
     isSpace[' '] = isSpace['\r'] = isSpace['\n'] = isSpace['\t'] = true; isLineSep['\r']
     = isLineSep['\n'] = true; }  ←空白文字の判定を行うメソッド
```

3つのソースコードの間でメイン処理部に大きなコーディングスタイルの違いは見られないが，テンプレート部のコーディングスタイルに違いが見られる．

図9と図10よりグループ18以上は1行あたりのイコールとスペースの最大数が18未満のグループに対して多いことがわかる．ソースコード18の20〜22行目とソースコード19の17行目にはいずれも空白文字の判定を行うテンプレートが含まれているため，1行に含まれるイコールとスペースの最大数が多くなっている．しかし，ソースコード17には空白文字の判定を行うテンプレートが含まれず，複数条件の判定をする処理が含まれていないため，最大数は多くならなかった．

グループが上がるほど空白文字の判定を含むテンプレート部を持つ開発者が増えることから，18未満のグループに比べてが1行あたりのイコールとスペースの最大数が多くなったと考えられる．

図11, 図12, 図13よりグループ18以上は1行あたりの識別子とピリオドの最大数が18未満のグループに対して多く，1行の最大長が長いことがわかる．図9, 図10に示したとおり，max_equalはグループ1〜18，max_spaceはグループ1〜17に対してグループ20の値が高い．しかし，max_identifiersは11以降のグループの値に対してグループ20の値に差がない．max_line_lenも13以降のグループの値に対して，max_periodも14以降のグループの値に対してグループ20の値に差がなかった．すなわち，グループ11〜14の中級者に対して，グループ20の上級者のmax_identifiers, max_line_len, max_periodの値に差はなかった．

ソースコード17には1行で複雑な処理をするテンプレートが含まれていないため，1行に含まれる識別子・ピリオドの最大数は多くならず，1行の最大長も長くない．しかし，ソースコード19には改行がなく1行で行う処理が多いテンプレート部が存在するため，値が大きくなった．一方，ソースコード18には24行目のように1行で書かざるを得ない処理が含まれているため値が大きくなったが，複数の処理を1行にまとめているわけではないので，ソースコード19ほど値が大きくならなかった．

グループが上がるほどテンプレート部を持つ開発者が増え，特にグループ18,

19の開発者は改行がなく1行で行う処理が多いテンプレート部を持つことから，このような結果になったと考えられる．

6 おわりに

本研究では，初級者が書くソースコードの可読性を評価し上級者との違いを示すことで，初級者が可読性を意識したコーディングを習得できると考えた．そのために，コーディングスタイルを定量的に計測するメトリクスを算出し，その値が初級者と上級者でどのように異なるのかを分析した．メトリクスごとに一元配置分散分析を行った結果，avg_blank_lines 以外の全てのメトリクスでグループごとの平均値に有意差（有意水準 1%）があり，開発者の能力によりコーディングスタイルに差があることがわかった．そこで，メトリクスごとに Ryan の方法による多重比較を行い，どのグループ間で平均値に有意な差があるのかを確認し，有意差のあったグループの傾向でメトリクスを分類し分析した．

実験の結果，グループ 20 の上級者は他の開発者に比べて 1 行あたりの識別子の数が少ないことがわかった．グループ 20 の開発者は，処理を複数行に分け 1 行の処理を簡潔にするコーディングスタイルであるためと考えられる．また，グループ 2, 5 の初級者は他の開発者に比べて 1 行あたりのインデントの数が多いことがわかった．グループ 2, 5 の開発者が作成するプログラムは，他グループの開発者が作成するプログラムに比べアルゴリズムが複雑である傾向があるためと考えられる．そして，グループ 18, 19 の上級者は他の開発者に比べて 1 行あたりのカッコの平均の数や最大の数などが多いことがわかった．グループ 18, 19 の開発者が作成するソースコードには，改行がなく 1 行で行う処理が多いテンプレート部が存在することが多い傾向にあるためと考えられる．これらのコーディングスタイルの特徴を用いることで，開発者の能力判定をプログラムの出力が正しいかだけでなく，ソースコードの可読性が高いかという観点でも行うことができる．可読性を大学などで利用が進みつつある自動採点システムで評価することが可能になれば，可読性を重要視したプログラミング教育を行うことができるため，教育の質の向上に有用だと考えられる．

今後の課題として，メイン処理部とテンプレート部を区別する手法の提案があげられる．これにより，問題の処理に必要な部分のみを評価できるため，ソースコードの性質を正確に理解できる．また，変数名の付け方や，コメントの有無や内容などによる可読性の定量化もあげられる．可読性を評価できる要素が増えれば，初級者は可読性を意識したコーディングをより多く習得できると考えられる．

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秀剛准教授には，研究を進めるにあたってご指導いただきました．心から感謝申し上げます．

また，査読教員である松尾賢一教授をはじめ，山口智浩教授，内田眞司教授には，貴重な意見や指摘をいただきました．誠に感謝しております．

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] 佐々木唯, 肥後芳樹, 楠本真二: プログラム文の並べ替えに基づくソースコードの可読性向上の試み, 情報処理学会論文誌, Vol.55, No2, pp.939-946 (2014).

[4] Buse, R.P.L. and Weimer, W.R.: Learning a Metric for Code Readability, IEEE Trans. Softw. Eng., Vol36, No.4, pp.546-558 (2010).

[5] 鈴木翔, 阿萬裕久, 川原稔: メソッド名の長さと構成に着目したソースコード品質に関する定量的調査, 研究報告ソフトウェア工学（SE）, Vol.2016-SE-194, No.6, 1-8 (2016).

[6] 阿萬裕久: オープンソースソフトウェアにおけるコメント記述およびコメントアウトとフォールト潜在との関係に関する定量分析, 情報処理学会論文誌, Vol.53, No.2, 612-621 (2012).