社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ソースコードの“自然さ”を用いたリファクタリング評価手法の検討

有馬　　諒†　肥後　芳樹†　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{r-arima,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　リファクタリングとはソフトウェアの外部的振る舞いを保ちながら内部構造を改善する作業と定義されており，ソフトウェアの保守性を向上させるために重要な作業である．しかし，Extract method と Inline method のように対立するリファクタリング手法も存在するため，どのようなリファクタリングを行うかは開発者の経験や勘に基づいて評価される場合が多い．そこで本研究では，開発者の経験や勘によらない定量的なリファクタリングの評価として，ソースコードの “自然さ” を用いる手法を提案する．自然さとは自然言語処理の手法である言語モデルを用いて，ソースコードがどの程度自然かを数値で表したものである．本研究では提案手法の評価のために，実際のソースコードに対して行われたリファクタリングに対して本手法を適用した．その結果，28 個のリファクタリングのうち，19 個のリファクタリングにおいて良いリファクタリングであると評価された．そのため，開発者によって行われたリファクタリングと，提案手法によって良いと評価されたリファクタリングが対応していると考えられ，提案手法がリファクタリングの評価に有効であることを確認した．

キーワード　自然さ，リファクタリング

1. ま え が き

近年，ソフトウェアの大規模化，複雑化によってソフトウェア保守の作業量は増加しており，ソフトウェアの保守性を保つためにもソースコードの品質は重要である．しかし，機能の追加や変更，バグ修正などによって，ソースコードには変更を加えられ続けるため，ソースコードの品質は徐々に低下していく [1]．このような品質の低下を抑えるために，リファクタリングという作業が行われる．リファクタリングとは，ソフトウェアの外部的な振る舞いを変えずに内部構造を改善する作業である．リファクタリングによって，ソースコードの品質を向上させることができる．Fowler らは，リファクタリングすべきソースコードの状態とそのリファクタリング方法についてまとめている [2]．しかし，ソースコード中のどの部分にどのようなリファクタリングを適用するかは，開発者の経験や勘によるものが大きく，リファクタリングを行うことを難しくしている．

開発者の経験や勘が必要なリファクタリングの例として，Extract Method と Inline Method がある．Extract Method リファクタリングとは，ソースコード中の処理をメソッドとして切り出すリファクタリングである．このリファクタリングによって，重複した処理を再利用できること，意味のある処理の単位を 1 つのメソッドとしてまとめて名前をつけることでソースコードの可読性が向上することなどの利点があげられる．また Inline Method リファクタリングとは，複数のメソッドに分解された処理を 1 つのメソッドに統合するリファクタリングである．必要以上に細かく分割されたメソッドを統合することで可読性が向上するといわれている．このように Extract Method リファクタリングと Inline Method リファクタリングは互いに対立したリファクタリングであり，どちらを適用するかはメソッドの規模や処理内容などから開発者が判断しなければならない．このような状況において，リファクタリングの評価を何らかの方法で数値化しどちらの状態がよりよいか比較できるようになれば，リファクタリングを行う開発者の支援になると著者らは考える．

そこで本研究では，“自然さ” を用いてリファクタリングを評価する手法を提案する．自然さとは，自然言語処理の手法である言語モデルを用いて求められる指標であり，単語列がある言語としてどれだけもっともらしいかを数値で表したものである．近年，自然さをソースコードに応用する研究が盛んに行われており，コード補完 [3] や，バグ検出 [4] などで成果が報告されている．提案手法ではリファクタリングを数値によって評価することができるため，複数のリファクタリングの候補が存在する場合，提案手法を用いてどのリファクタリングが最も良いかを比較することができる．

提案手法の評価のために，実際のソースコードに対して行われたリファクタリングに対して提案手法を適用した．その結果，28 個のリファクタリングのうち，19 個のリファクタリングにおいて良いリファクタリングであると判定された．そのため多くのリファクタリングにおいて，開発者によって行

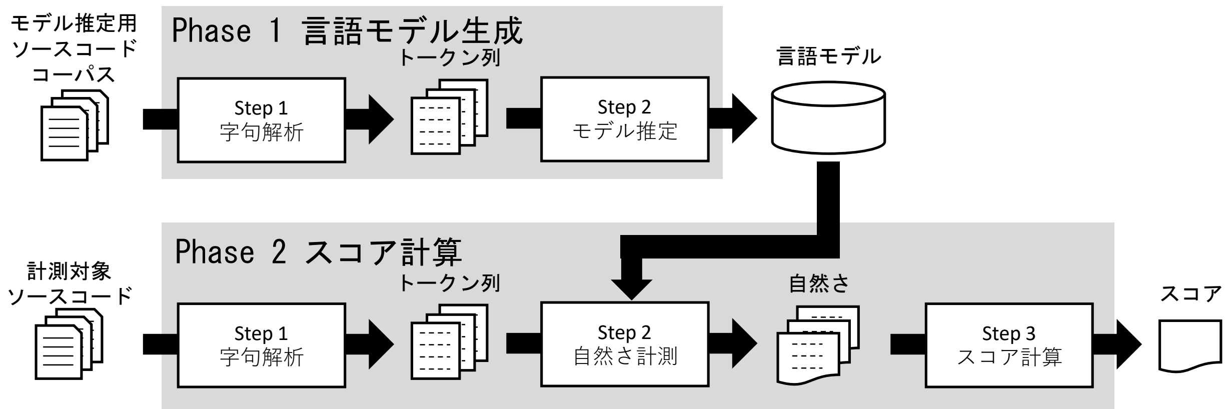

図1　提案手法の概要

われたリファクタリングと，提案手法によって良いと評価されたリファクタリングが対応していると考えられる．

以降，2章では本研究の背景として言語モデルと自然さについて述べる．3章では提案手法について述べる．4章では提案手法の評価のために行った実験について述べる．5章では妥当性の脅威について述べ，最後に6章ではまとめと今後の課題について述べる．

2. 背　　景

2.1 言語モデルと自然さ

言語モデルとは単語列に対してその生成確率を割り当てる確率モデルである．言語モデルを用いることで，単語列の“自然さ”を数値で表すことができる．単語列 $S = w_1 w_2 \cdots w_m$ が与えられたときその生成確率は，各単語の条件付確率の積を用いて次のように表される．

$$P(S) = P(w_1)\prod_{i=2}^{m} P(w_i|w_1, \cdots, w_{i-1}) \tag{1}$$

ここで $P(w_i|w_1, \cdots, w_{i-1})$ は，単語列 $w_1, \cdots, w_{i-1}$ の次の単語が w_i である確率である．$P(w_i|w_1, \cdots, w_{i-1})$ は，単語列 $w_1, \cdots, w_{i-1}$ の組み合わせが膨大になるため，直接求めることは現実的ではない．そこで，以下のように直前の $n-1$ 個の単語から次の単語の確率を求める $n-gram$ 言語モデルを考える．

$$P(w_i|w_1, \cdots, t_{w-1}) \simeq P(w_i|w_{i-n+1}, \cdots, w_{i-1}) \tag{2}$$

$P(w_i|w_{i-n+1}, \cdots, w_{i-1})$ はコーパス中の単語列の頻度から，最尤推定によって以下のように求められる．

$$P(w_i|w_{i-n+1}, \cdots, w_{i-1}) = \frac{C(w_{i-n+1}, \cdots, w_{i-1}, w_i)}{C(w_{i-n+1}, \cdots, w_{i-1})} \tag{3}$$

ここで $C(w_{i-n+1}, \cdots, w_{i-1})$ はコーパス中で単語列 $w_{i-n+1}, \cdots, w_{i-1}$ が出現した回数を表す．

$P(w_i|w_{i-n+1}, \cdots, w_{i-1})$ は，非常に小さな値となることが多いためそのままでは扱いづらい．そこで，本研究では単語 w_i の自然さ $N(w_i)$ を，以下のように単語の確率の対数とする．

$$N(w_i) = \log P(w_i|w_{i-n+1}, \cdots, w_{i-1}) \tag{4}$$

従って，単語列 $S = w_1 w_2 \cdots w_m$ の自然さ $N(S)$ は以下のようになる．

$$N(S) = \log P(w_1) + \sum_{i=2}^{m} \log P(w_i|w_{i-n+1}, \cdots, w_{i-1}) \tag{5}$$

言語モデルや自然さは自然言語処理において，機械翻訳や音声認識などで用いられている．

2.2 ソースコードにおける自然さ

Hindle らは，自然言語処理の手法である言語モデルをソースコードに適用し，言語モデルがソースコードに対しても有効であることを示した [3].

Allamanis らは，言語モデルを用いて最適な識別子名の推薦する手法を提案した [5].

Ray らは，バグを含むソースコードと自然さについて調査を行った [4]．この研究では，ソースコードにおいて，バグを含む行は自然さが低い傾向があることを示し，またバグ修正によって自然さが向上する傾向があることを示した．これらの事実から自然さをバグ検出に応用する手法を提案した．

本研究では，ソースコードの自然さをリファクタリングの評価に用いる手法を提案する．

3. 提 案 手 法

本研究では，ソースコードの自然さを用いてリファクタリングを評価する手法を提案する．提案手法に用いるアイデアは以下のとおりである．開発者は，複雑で理解しづらく可読性の低いソースコードが，単純で理解しやすく可読性の高いソースコードとなるようにリファクタリングを行う．自然なソースコードというのは，学習に用いたソースコードと似た書き方であるソースコードであり，学習に用いたソースコードの可読性が高いならば，自然なソースコードというのは可読性の高いソースコードとなる．このことから，リファクタリングと自然さの関係について以下の仮説をたてた．

学習に用いたソースコードが単純で理解しやすく可読性の高いソースコードであれば，開発者がリファクタリングを行うことによって自然さの低い行が減少する．

本研究ではこの仮説をもとに，自然さを用いてリファクタリングを評価する手法を提案する．提案手法の概要を図1に示

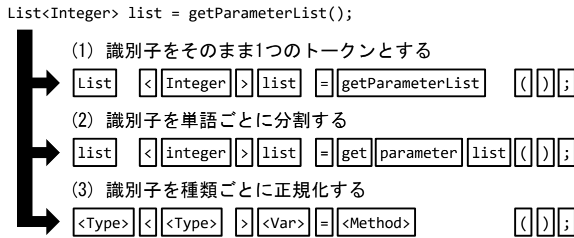

図 2　字句解析の実行例

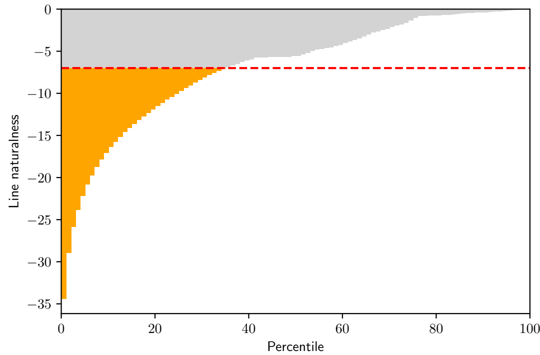

図 3　スコア計算の例

す．提案手法の入力は，言語モデル推定用ソースコードコーパスと，計測対象ソースコードであり，出力は計測対象ソースコードのスコアである．このスコアはより自然なソースコードであるほど減少するため，リファクタリングによって提案手法のスコアが減少するならば，良いリファクタリングであると評価することができる．

提案手法は以下の 2 つの Phase からなる．

Phase 1　言語モデル生成

Phase 2　スコア計算

以降，各 Phase について説明する．

3.1　言語モデル生成

Phase 1 では学習用ソースコードコーパスから，言語モデルを構築する．この Phase は以下の 2 つの Step からなる．

Step 1　字句解析

Step 2　モデル推定

Step 1 では，ソースコードを解析し，トークンの並びに分解する．本研究ではソースコードのうち，メソッドの本体部分のみを用いた．その際問題となるのは，識別子名の扱いである．ソースコード中のクラス名やメソッド名，変数名などの識別子名は開発者が自由に決めることができ，多くの場合複数の英単語を組み合わせて 1 つの識別子名とする．そのため，ソースコード中に含まれる識別子名の種類が膨大になり，うまくモデルを推定できない可能性がある [6]．また，似た単語から構成される識別子も全く別の識別子としてみなしてしまうという問題もある．そこで，以下の 3 つの戦略で識別子名を扱う．

（1） 識別子をそのまま 1 つのトークンとする．

（2） 識別子を単語ごとに分割する．

（3） 識別子を種類ごとに正規化する．

```
public void add(String left, String right){                                  自然さ
  int leftValue = 0;                                                          -4.4
  for(int i = 0; i < left.length(); i++){                                     -8.7
    leftValue *= 16;                                                          -6.6
    leftValue += 'A' <= left.charAt(i) && left.charAt(i) <=                  -73.6
      'F' ? left.charAt(i) - 'A' + 10 : left.charAt(i) - '0';
  }                                                                           -0.8

  int rightValue = 0;                                                         -3.6
  for(int i = 0; i < right.length(); i++){                                    -8.4
    rightValue *= 16;                                                         -6.6
    rightValue += 'A' <= right.charAt(i) && right.charAt(i) <=               -73.8
      'F' ? right.charAt(i) - 'A' + 10: right.charAt(i) - '0';
  }                                                                           -0.8

  System.out.println(leftValue + " " +  rightValue);                          -7.9
}
```

図 4　リファクタリング対象ソースコード

```
public void add(String left, String right){                                  自然さ
  int leftValue = convertHex(left);                                           -9.3
  int rightValue = convertHex(right);                                         -7.8
  System.out.println(leftValue + " " +  rightValue);                          -8.1
}

public int convertHex(String str){
  int value = 0;                                                              -5.3
  for(int i = 0; i < str.length(); i++){                                      -5.8
    value *= 16;                                                              -9.1
    value += 'A' <= str.charAt(i) && str.charAt(i) <= 'F' ?                  -60.6
      str.charAt(i) - 'A' + 10: str.charAt(i) - '0';
  }                                                                           -0.8
  return value;                                                               -3.5
}
```

図 5　メソッド切り出し後のソースコード

```
public void add(String left, String right){                                  自然さ
  int leftValue = convertHex(left);                                           -9.3
  int rightValue = convertHex(right);                                         -7.8
  System.out.println(leftValue + " " +  rightValue);                          -8.1
}

public int convertHex(String str){
  int value = 0;                                                              -5.3
  for(int i = 0; i < str.length(); i++){                                      -5.8
    value *= 16;                                                              -9.1
    char c = str.charAt(i);                                                   -6.8
    if('A' <= c && c <= 'F'){                                                 -7.5
      value += c - 'A' + 10;                                                 -12.3
    }else{                                                                    -1.5
      value += c - '0';                                                      -11.8
    }                                                                         -0.8
  }                                                                           -0.8
  return value;                                                               -3.2
}
```

図 6　一時変数の導入後のソースコード

字句解析の例を図 2 に示す．

Step 2 では，字句解析に得られたトークン列から，言語モデルを推定する．本手法では，2 種類の言語モデル推定器を用いた．1 つ目は KenLM [7] である．これはスムーシングに，自然言語処理において高い性能を示している modified Kneser-Ney smoothing [8] を用いた n-gram 言語モデルである．

2 つ目は Tu らの研究 [9] によって用いられた言語モデルである．ソースコードではローカル変数など局所的に頻度が高まるトークンが存在することが知られており，この特徴を考慮するためにキャッシュの仕組みを導入した n-gram 言語モデルである．

3.2　自然さ計測

Phese 2 では，Phese 1 で構築した言語モデルを用いて計測対象ソースコードの自然さを計算し，この自然さをもとにスコアを計算する．この Phese は以下の 3 つの Step からなる．

Step 1　字句解析

Step 2　自然さ計測
Step 3　スコア計算

Step 1 では，Phase 1 と同様に計測対象ソースコードの字句解析を行う．ここでは，Phase 1 の際の戦略と同じものを用いる必要がある．Step 2 では，Step 1 で得られたトークン列に対して，Phase 1 で生成した言語モデルを用いて行ごとの自然さを計測する．Step 3 では，計測した自然さをもとに，以下のようにスコアを計算する．

$$score = \frac{1}{M}\sum_{i=0}^{M} max(0, threshold - N(L_i)) \quad (6)$$

ここで，M は計測対象ソースコードの行数，$threshold$ は閾値，$N(L_i)$ は i 番目の行の自然さを表す．本研究では，閾値に学習用ソースコードコーパスにおける自然さの分布の中央値を用いる．このスコアが低いほど自然なソースコードである．

このスコア計算の意味を図 3 を用いて説明する．図 3 は，各行の自然さを低い順に並び替え，x 軸 0 から 100 の間にプロットしたグラフである．中央値を破線としたとき提案手法のスコアはこの破線より下の部分の面積であり，面積が小さいほど自然なソースコードとなる．

こうして得られたスコアをリファクタリングの前後で比較しスコアが減少しているならば，このリファクタリングは良いリファクタリングであると提案手法は評価する．

3.3 適 用 例

リファクタリングによるスコアの変化の例を図 4 のソースコードを用いて示す．図において，add メソッドは 16 進数で表された 2 つの文字列の加算結果を出力するメソッドである．学習用ソースコードコーパスにおける自然さの中央値が −5.7 であったため，このソースコードのスコアは 145.6 であった．

まず，16 進数文字列を数値に変換する処理が重複しているため，この処理をメソッドとして切り出すリファクタリングを行った．この結果を図 5 に示す．このソースコードのスコアは 66.6 であり，リファクタリングによって元のソースコードよりもスコアが減少した．

さらに，16 進数の各桁を処理する行は，1 つの行で多くの処理を行っているため，この行を分解するリファクタリングを行った．この結果を図 6 に示す．このソースコードのスコアは 27.3 であり，図 5 のソースコードよりもさらに自然なソースコードとなったといえる．

4. 評 価 実 験

本章では，提案手法の評価のために行った実験について述べる．この実験では，オープンソースソフトウェアにおいて実際に行われたリファクタリングに対して提案手法を適用することで，提案手法で用いた仮説「開発者は，自然さの低い行が減少するようにリファクタリングを行う」が成り立ち，開発者によって行われたリファクタリングと，提案手法によって良いと評価されたリファクタリングが一致しているかを確認した．

表 1　データセットの概要

プロジェクト数	84
ファイル数	66,724
LOC	11,545,556

4.1 実 験 方 法

学習用ソースコードコーパスには，Higo らによって作成されたデータセット [10] を用いた．これは Apache Software Foundation(注1) の Java プロジェクトから作成されたデータセットである．データセットの概要を表 1 に示す．

リファクタリングの評価対象には，JUnit4(注2) を用いた．これは Java 用の単体テスト自動化フレームワークであり，JUnit4 自体も Java によって実装されている．JUnit4 は GitHub によって管理されており，GitHub を用いた開発者相互によるソースコードレビューによってソースコードの品質が担保されている．

本研究では JUnit4 リポジトリから，リファクタリングのみが行われたコミットを抽出し，それを評価対象として用いた．まず，コミットメッセージにリファクタリングに関連があると思われるキーワードを含むコミットを抽出した．本実験ではキーワードとして *refactor*，*clean* を用いた．次に抽出されたコミットを目視で確認し，変更内容がリファクタリングのみであり，処理の内容を変えないコミットのみを抽出した．こうして抽出した 28 コミットを評価対象として用いた．対象リポジトリのメトリクスを表 2 に示す．

抽出した各コミットについて，変更前のソースコードと，変更後のソースコードのそれぞれに対して提案手法を適用し，それぞれのスコアを求めた．このときスコア計算に必要な閾値は，学習用ソースコードコーパスに対して交差検定を行って行ごとの自然さの分布を求め，その中央値を用いた．こうして求めたスコアが変更の前後で増加しているか，減少しているかを確認した．

4.2 結 果

実験結果を，表 3 に示す．最も性能の良い言語モデルと識別子処理の組み合わせでの場合で，28 個のリファクタリングのうち 19 個で提案手法によるスコアが減少した．この結果から，多くのリファクタリングで開発者によって行われたリファクタリングと，提案手法によって良いと評価されたリファクタリングが対応しているといえる．変化なしとなった 3 つのリファクタリングは，フィールド変数の修飾子の変更やメソッ

表 2　JUnit リポジトリの詳細

期間	2000/12/03 - 2017/10/16
最新コミットでのファイル数	449
最新コミットでの LOC	43161
全コミット数	2195
キーワードを含むコミット数	354
目視確認による リファクタリングコミット数	28

(注1)：http://www.apache.org/
(注2)：https://github.com/junit-team/junit4

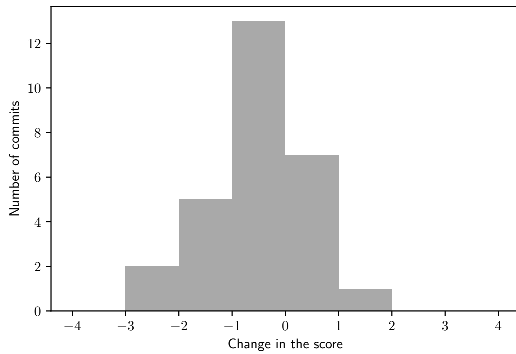

図 7　リファクタリングによるスコアの変化量

```
                                                                     自然さ
+ private List getAnnotatedFieldsByParameter() {
+   return getTestClass().getAnnotatedFields(Parameter.class);      -50.5
+ }

+ private boolean fieldsAreAnnotated() {
+   return !getAnnotatedFieldsByParameter().isEmpty();              -44.2
+ }
```

(a) 切り出されたメソッド

```
                                                                     自然さ
 @Override
  protected void validateConstructor(List<Throwable> errors) {
    validateOnlyOneConstructor(errors);                              -19.2
-   List<FrameworkField> annotatedFieldsByParameter =                -84.4
      getTestClass().getAnnotatedFields(Parameter.class);
-   if (annotatedFieldsByParameter.size() > 0) {                     -62.3
+   if (fieldsAreAnnotated()) {                                      -32.1
        validateZeroArgConstructor(errors);                          -2.9
    }
  }

  @Override
  protected void validateFields(List<Throwable> errors) {
    super.validateFields(errors);                                    -37.1
-   List<FrameworkField> annotatedFieldsByParameter =                -90.8
      getTestClass().getAnnotatedFields(Parameter.class);
-   if (annotatedFieldsByParameter.size() > 0) {                     -60.3
+   if (fieldsAreAnnotated()) {                                      -36.7
+     List<FrameworkField> annotatedFieldsByParameter =              -56.7
      getAnnotatedFieldsByParameter();
```

(b) メソッド呼び出しへの変更

図 8　スコアが減少した例

ドの移動など，提案手法のようなメソッド本体のみを考慮する手法では評価できないリファクタリングである．

最も性能の良い言語モデルと識別子処理の組み合わせの 1 つである，Cache モデルと識別子そのままの場合におけるすべてのコミットでの結果を表 4 に示す．✓ はスコアが減少し，提案手法がよいリファクタリングと評価したコミットである．また，このときのリファクタリング前後でのスコアの差をヒ

表 3　実 験 結 果

言語モデル	識別子	増加	変化なし	減少
KenLM	そのまま	9	3	16
KenLM	分割	9	3	16
KenLM	正規化	8	3	17
Cache	そのまま	6	3	19
Cache	分割	6	3	19
Cache	正規化	8	3	17

```
                                                                     自然さ
+ private Collection<T> getFilteredChildrenWithoutIgnores(
+           final RunNotifier notifier) {
+   final Collection<T> filteredChildren                             -60.1
      = getFilteredChildren();
+   Collection<T> filteredChildrenCopy                               -55.3
      = new ArrayList<T>(filteredChildren);
+
+   for (T child : filteredChildren) {                               -47.6
+     if (isIgnoredMethod(child)) {                                  -43.7
+       Description childDescription= describeChild(child);         -29.2
+       notifier.fireTestIgnored(childDescription);                  -29.6
+       filteredChildrenCopy.remove(child);                          -31.4
+     }                                                              -14.1
+
+   }                                                                 -6.2
+
+   return Collections                                               -20.6
+           .unmodifiableCollection(filteredChildrenCopy);           -24.5
+
+ }
```

(a) 切り出されたメソッド

```
                                                                     自然さ
 protected Statement childrenInvoker(final RunNotifier notifier) {
-   final Collection<T> filteredChildren= getFilteredChildren();     -61.6
-   Collection<T> filteredChildrenCopy=                              -56.0
      new ArrayList<T>(filteredChildren);
-
-   for (T child : filteredChildren) {                               -48.2
-     if (isIgnoredMethod(child)) {                                  -44.2
-       Description childDescription= describeChild(child);         -29.4
-       notifier.fireTestIgnored(childDescription);                  -29.9
-       filteredChildrenCopy.remove(child);                          -31.8
-     }                                                              -14.3
-
-   }                                                                 -6.2
-
-   final Collection<T> filteredChildrenWithoutIgnores=Collections   -23.7
-         .unmodifiableCollection(filteredChildrenCopy);             -24.7
+   final Collection<T> filteredChildrenWithoutIgnores=              -65.2
      getFilteredChildrenWithoutIgnores(notifier);

    if (filteredChildrenWithoutIgnores.isEmpty()) {                  -53.0
      return new EmptyStatement();                                   -29.0
    }                                                                -12.8

    return new Statement() {                                         -31.2
      @Override
      public void evaluate() {
        runChildren(notifier, filteredChildrenWithoutIgnores);       -31.2
      }
    };                                                               -14.4
  }
```

(b) メソッド呼び出しへの変更

図 9　スコアが増加した例

ストグラムで表したものを図 7 に示す．このヒストグラムからも，リファクタリングによってスコアが減少する傾向があることがわかる．

スコアが減少した例として，2012/8/20 15:47 のコミット `#0215c66` を図 8 に示す．このコミットでは Extract Method リファクタリングが行われており，図 8(a) に示す 2 つのメソッドが切り出されている．このとき，行の自然さ-50.3 は交差検定による分布中では下位 14%に相当し，-44.2 は下位 18%に相当する．切り出した 2 つのメソッドを使用するように変更された部分を図 8(b) に示す．この変更では，自然さの低かった行がメソッド呼び出しに変更されることで自然さが向上していることがわかる．このように，自然さが低く繰り返し出現する行をメソッドとしてまとめることで自然さの低い行が削減されたためスコアが減少したと考えられる．

スコアが増加した例として，2013/10/21 01:58 のコミット `#2240984` を図 9 に示す．このコミットにおいても Extract Method リファクタリングが行われており，図 9(a) に示す `getFilteredChildrenWithoutIgnores` メソッドが切り出され

ている．切り出したメソッドを使用するように変更された部分を図 9(b) に示す．この変更では，Extract Method リファクタリングによって新たなメソッドが追加されたものの，ソースコード中に存在する行自体はほぼ変わっておらず，新たなメソッド呼び出しが追加された分だけスコアが増加した．提案手法は n-gram によって行単位の自然さを計測しているため，各行が自然か，そうでないかは判定できるが，それら行の並びが自然かそうでないかはほぼ考慮することができない．そのため図 9 のようなリファクタリングを評価することは難しい．

5. 妥当性の脅威

5.1 学習用ソースコードの品質

言語モデルを用いた自然さは，出現頻度が高いパターンであるほどより自然になるため，学習用ソースコードに保守性の低いソースコードが多く含まれている場合，品質の低いソースコードに対して自然であると出力してしまう可能性がある．しかし今回学習用コーパスに用いたソースコードは，Apache Software Foundation によって管理されているオープンソースプロジェクトであるため，品質は十分であると考えられる．

5.2 評価用データ

評価実験では 1 つのプロジェクトの 28 コミットしか評価されておらず，プロジェクトによる偏りが存在する可能性がある．また，リファクタリングが行われたかどうかは目視によって判別しているため誤りが含まれる可能性がある．

表 4 すべてのコミットでの結果．✓ は良いリファクタリングと評価されたコミットである．

コミット ID	リファクタリングの内容	スコアの変化	評価
#23793cd	Extract class	0.46	
#a7c4d03	Extract class	-1.76	✓
#7a2b046	Extract class	-0.62	✓
#fd1ef3c	Extract method	-0.21	✓
#dbe7711	Extract method	-0.49	✓
#2240984	Extract method	0.07	
#862f41c	Extract method	0.91	
#5976b1d	Extract method	-0.15	✓
#0215c66	Extract method	-2.78	✓
#24cbcbc	Extract method	-2.19	✓
#467dd07	Extract method	-0.26	✓
#e48f6d4	Extract method	-1.09	✓
#fe5d86e	Inline method	-1.01	✓
#6838ac0	Inline method	0.49	
#0030e51	Inline method	1.02	
#ce9bc58	Move method	0.00	
#f1f4fe2	Move method	-0.72	✓
#4c1758d	アルゴリズム変更	-0.38	✓
#66bfb24	アルゴリズム変更	-1.14	✓
#df016dc	アルゴリズム変更	-0.24	✓
#17a2f11	不要な処理の除去	-1.15	✓
#9a0aec8	不要な継承の除去	0.00	
#a30e87b	変数に final を付与	0.00	
#db8d580	コードクローンの除去	-0.49	✓
#e77e1c4	既存メソッドの再利用	-0.23	✓
#759061a	Strategy パターンの導入	-0.04	✓
#7f2569f	Pull up method	-0.46	
#d064212	Rename method and class	-0.50	✓

6. あとがき

本研究ではリファクタリング支援手法開発の第一歩として，ソースコードの自然さを用いたリファクタリングの評価手法を提案した．提案手法の評価実験では，オープンソースプロジェクトに対して行われた 28 個のリファクタリングのうち，19 個のリファクタリングにおいて良いリファクタリングであると評価された．そのため開発者によって行われたリファクタリングと，提案手法によって良いと評価されたリファクタリングが対応していると考えられ，提案手法がリファクタリングの評価に有効であることを確認した．

今後の課題としてソースコードの特徴をより反映し，高精度なリファクタリング評価手法を開発することがあげられる．自然言語処理の分野では，Deep Learning による技術が大きく進歩しており，これらの技術を取り入れる方法を検討している．

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：JP25220003）の助成を得て行われた．

文献

[1] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, “Does code decay? assessing the evidence from change management data,” IEEE Transactions on Software Engineering, vol.27, no.1, pp.1–12, 2001.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.

[3] A. Hindle, E.T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the Naturalness of Software,” Proceedings of the 34th International Conference on Software Engineering, pp.837–847, 2012.

[4] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, “On the “Naturalness” of Buggy Code,” Proceedings of the 38th International Conference on Software Engineering, pp.428–439, 2016.

[5] M. Allamanis, E.T. Barr, C. Bird, and C. Sutton, “Learning Natural Coding Conventions,” Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.281–293, 2014.

[6] V. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?,” Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp.763–773, 2017.

[7] K. Heafield, “KenLM: faster and smaller language model queries,” Proceedings of Sixth Workshop on Statistical Machine Translation, pp.187–197, 2011.

[8] S.F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” Computer Speech & Language, vol.13, no.4, pp.359–394, 1999.

[9] Z. Tu, Z. Su, and P. Devanbu, “On the Localness of Software,” Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.269–280, 2014.

[10] Y. Higo and S. Kusumoto, “How should we measure functional sameness from program source code? an exploratory study on java methods,” Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.294–305, 2014.