B-004

Java プログラミングにおけるリファクタリング支援ツールに関する研究
Refactoring Support Tool for Java Programming

菊池 禎人† 蓬莱 尚幸†
Yoshihito Kikuchi Hisayuki Horai

1. はじめに

ソフトウェア開発において、適切に設計されたアプリケーションであっても、時間や環境の変化、顧客やユーザの要望などのさまざまな理由により、使用するプログラミング言語に関わらず変更は行われ続ける[1]。また、変更が加えられる際にソースコードが柔軟なものでない場合、機能的に小さい変更であっても実際に変更を加える箇所が多くなってしまうことがある。こうした変更を繰り返していくと予期せぬバグやコーディングの重複による修正時間の増加の原因となってしまう。

このようなコードの劣化を防ぐための手法としてリファクタリングが挙げられる。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保ちつつ、ソースコードを分かりやすく改善することである。リファクタリングを行うことで、コードを読みやすく、変更に対して柔軟な設計にすることができる。

しかし、リファクタリングには厳密な定義がなく、どのようにリファクタリングを行うのか、行うタイミングはいつなのか、それらを理解するには多くの経験と知識を要する[2]。そのため、経験の少ないプログラマにとってリファクタリングを行うことは容易ではない。

リファクタリングを行う指標は、「不吉な臭い」と呼ばれている。不吉な臭いとは、ソースコード内の重複したコードや長すぎるメソッド、巨大なクラスなど、リファクタリングが必要である可能性が考えられる兆候や雰囲気のことを言う[3]。

本研究では、ソースコード内の不吉な臭いを検出し、プログラミング経験の浅いプログラマに対して、リファクタリング箇所の検出支援を目的としたツールの作成を行う。臭いの検出は print 文中の文字列の類似性をもとに行う。動的計画法(DP)を利用した配列アラインメントを用いて文字列の類似度を算出し、ノードを文字列、エッジを算出した類似度とした重み付きグラフを作成・分類する。

†茨城工業高等専門学校専攻科情報工学コース
Advanced Course of Information Engineering, National Institute of Technology, Ibaraki College

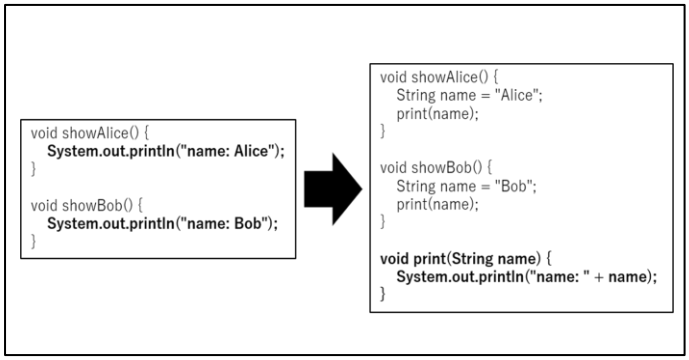

図 1 メソッド抽出例

2. 不吉な臭い

不吉な臭いにはさまざまなものがある。例えば、以下のようなものが挙げられる：

・複数箇所に存在する重複したコード
・長すぎるメソッド
・あまりに大きな仕事をしている巨大なクラス
・ポリモーフィズムに置き換え可能な switch 文

本研究では、リファクタリングの支援を行うために、複数箇所に存在する重複したコードを臭いとして検出する手法を考案し、支援ツールの機能として実装する。

3. 提案手法

ソースコードから臭いとされるような要素を検出する。前述の重複したコードを改善するための手法として図 1 に例として示すようなメソッド抽出が提案されている。図 1 では、関数名だけでなく文字列データの一部が一致しているため、メソッドを抽出している。本研究では、このようにメソッド抽出が行えそうな重複したコードを臭いとして検出する。

3.1. コードの臭い検出

メソッド内の print 文はそのメソッドで得られた結果を表示するために使われることが多い。そこで、メソッド内に存在する print 文中の文字列の類似度からメソッド間の類似度を測れるのではないかと考えた。また、比較した文字列をノード、類似度をエッジとした重み付きグラフを作成し、ノード間の重みが大きいものをまとめることで、多数ある文字列の中で類似度が高い文字列の集合を得られると考えた。本研究では、メソッド内の print 文中の文字列を比較し、その類似度をもとに、重み付きグラフを作成・分類することで臭いを検出する。

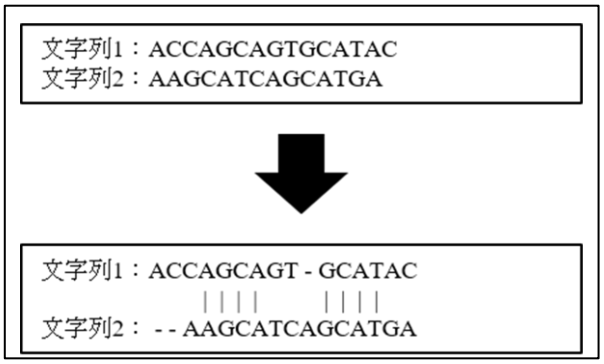

図 2　ローカルアラインメント例

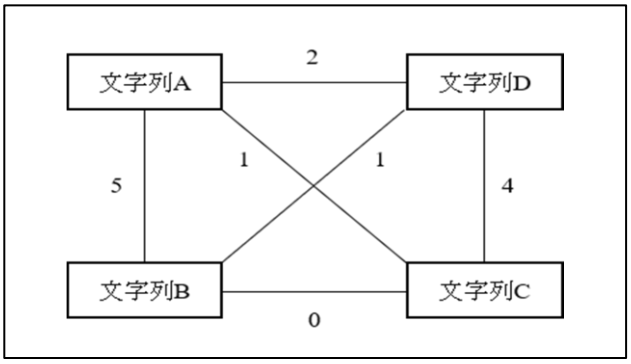

図 3　重み付きグラフ例

3.1.1. 類似度

print 文中の文字列の類似度を算出するために、DNA 配列解析において広く用いられている、DP を利用した配列アラインメントを用いる。これによって、出力する文字列が共通している、あるいは類似している部分が検出された場合はまとめられる可能性があるとして、臭いの候補とする。

3.1.2. 配列アラインメント

DP を用いたローカルアラインメントを使用して、文字列同士の類似している箇所を探す。ローカルアラインメントとは 2 つの文字列の間で局所的にアラインメントを行うことができる手法である。図 2 はローカルアラインメントを行った例であり、2 箇所アラインメントされていることがわかる。

比較する文字列は複数存在するので、マルチプルアラインメントを取る必要がある。本研究では、ペアワイズアラインメントを総当たりで行うことによって疑似的にマルチプルアラインメントを行う。ローカルアラインメントのアルゴリズムには Smith-Waterman アルゴリズムを用いる。

3.1.3. Smith-Waterman アルゴリズム

Smith-Waterman は、ローカルアラインメントを求めるアルゴリズムである。このアルゴリズムでは、アラインメントを求める際に、スコアが負にならないようにするため、以下に示すような制限条件がかけられている。この制限条件によってローカルアラインメントを求めることができる。

$$F(i,j) = \max \begin{cases} 0 \\ F(i-1,j-1) + s(x_i, y_j) \\ F(i-1,j) - d \\ F(i,j-1) - d \end{cases}$$

3.1.4. 重み付きグラフ

ローカルアラインメントを用いると、マッチした文字列の最大長をスコアとして得られる。本研究では、そのスコアを文字列間の類似度として用いる。この類似度を重みとして重み付きグラフを作成する。図 3 に簡単な重み付きグラフの例を示す。次に、閾値以下の重みを持ったエッジを切るなどしてグラフを分割する。例えば、閾値を 2 とした場合、「文字列 A, 文字列 B」「文字列 C, 文字列 D」をサブグラフとして分割することができる。分割して得られたサブグラフを類似した文字列の集合とする。それらの文字列がどのメソッド内のものなのかを判別することで、類似したメソッドを臭いとして検出する。

4. 今後の課題と解決方法

ローカルアラインメントによってスコアを得ることはできるが、比較する文字列の絶対的な長さによってスコアの信頼性にばらつきが生じてしまうため、そのスコアを直接重みとしてグラフ化してしまうと文字列の長さに依存した結果になってしまうので適切ではないと考えられる。

解決方法の一つとしては、ローカルアラインメントによって検出される一致した文字列すべてをベクトル化してそのベクトル同士の類似性を比較することで文字列の類似度を比較する方法が考えられる。

参考文献

[1] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates, “Head First デザインパターン 頭とからだで覚えるデザインパターンの基本”, (佐藤直生，木下哲也訳), オライリー・ジャパン, (2005)

[2] 兼光智子，肥後芳樹，楠本真二, “プログラム依存グラフを用いたリファクタリング候補の特定と可視化”, 電子情報通信学会技術研究報告：信学技報 110(336):61-66, (2010)

[3] 梅田政利, “不吉な匂い”, (2003), 「最終閲覧日：2019 年 6 月 20 日」, http://objectclub.jp/technicaldoc/refactoring/refact-smell