修士学位論文

題目

ソースコード理解性向上のための探索的リファクタリング手法

指導教員

楠本 真二 教授

報告者

谷門 照斗

平成 31 年 2 月 6 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻

平成 30 年度 修士学位論文

ソースコード理解性向上のための探索的リファクタリング手法

谷門 照斗

内容梗概

近年，ソフトウェアの大規模化・複雑化に伴い保守コストが増大している．保守作業全体の中で最も時間を占める作業はソースコード理解であると言われている．したがって，ソースコードの理解性を向上させることで保守作業コストを削減できると考えられる．ソースコードの理解性に関して，変数が宣言されてから参照されるまでの間に多くの処理を含む場合，理解するためのコストが増大するという報告がある．

そこで本研究では，ソースコードの理解性を向上させることを目的として，プログラム文を並べ替えることで変数の宣言と参照が近くなるようにリファクタリングする手法を提案する．すべての変数の宣言と参照の距離が最短となるような最適なリファクタリングを行うためには，膨大な計算量を必要とするため，提案手法ではリファクタリングの質と実行時間のバランスを利用者が調整することを可能にするために探索アルゴリズムを用いてリファクタリングを行う．探索アルゴリズムとは，試行錯誤を繰り返すことで多数の解の候補の中からより良い解を見つけるためのアルゴリズムであり，現実的に最適解を求めるのが困難な問題に対して近似解を得るために用いられる手法である．提案手法を実装し，オープンソースソフトウェアに対して適用したところ，開発者が理解しやすいリファクタリングを行うことができることを示した．

主な用語

ソースコード理解
リファクタリング
探索アルゴリズム

目次

図目次

表目次

1　はじめに

近年，ソフトウェアの大規模化・複雑化に伴い保守コストが増大している．ソフトウェアの保守性を保つために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が加え続けられるため，ソースコードの品質は徐々に低下していく．

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 [1]．リファクタリングによって，ソースコードの保守性や可読性を向上させられる一方，手動によるリファクタリングは誤ってバグを混入させてしまう恐れがあることが指摘されている [2]．

そこで，リファクタリング支援に関する様々な研究が行われおり，その中にリファクタリング自動化に関する研究がある [3]．リファクタリング自動化は一種の最適化問題と捉えることができる．つまり，ソースコードの保守性や可読性が目的関数であり，様々な変更を加えたソースコードが解の集合，ソースコードの変更方法が探索空間に相当する．この考え方に基づき，探索アルゴリズムを用いてリファクタリングを自動化する手法が多数存在する [3]．探索アルゴリズムを用いたリファクタリング自動化手法は，入力されたソフトウェアに対して適用可能な，より良いリファクタリング操作列を提案する．

保守作業全体の中で最も時間を占める作業はソースコード理解であると言われている [4, 5, 6]．したがって，ソースコードの理解性を向上させることで保守作業コストを削減できると考えられる．ソースコードの理解性に関して，変数が宣言されてから参照されるまでの間に多くの処理を含む場合，理解するためのコストが増大するという報告がある [7]．

この報告を受けて，佐々木らはプログラム文を並べ替えることで，変数の宣言と参照間の距離を短くリファクタリングする手法を提案している [8]．佐々木らの提案手法では，一定の条件を満たすすべての並べ替えパターンを生成した後，その中から変数の宣言・参照間距離が最小のものを出力するという戦略を取っている．こうすることで，変数の宣言・参照間距離が最小となる並びは得られるが，その反面実行時間は長くなってしまう．なぜなら，リファクタリングを行うプログラムの総行数を n としたときに，すべてのプログラム文の並べ替えの組み合わせを計算する時間計算量は $O(n!)$ となってしまい，n が大きくなった場合に現実的な時間で最適解を求めるのは困難なためである．

そこで，本研究では変数の宣言・参照間距離を短くするリファクタリングを探索的に行う手法を提案する．探索的にリファクタリングを行うことによって，最適解を得られる保証はなくなるが，得られるリファクタリングの質と実行時間のどちらを重視するかのバランスを利用者が調整することができる．利用者が実行時間が長くなることを厭わないのであれば，実行時間を長くして質の高いリファクタリングを行うこともでき，反対にリファクタリングの質を多少犠牲にして実行時間を短くすることもできる．提案手法を実装し，オープンソースソフトウェアに対して適用したところ，開発者に受け入れられ

るリファクタリングを行うことができた．

本論文では以降，2 章で準備としてソースコード理解やリファクタリング，ソフトウェアメトリクスについて説明する．3 章では本研究に関連する既存研究について述べる．4 章では提案手法を示し，5 章では提案手法の実装について説明する．6 章では提案手法の評価実験について述べる．7 章では評価実験に対する妥当性への脅威について述べる．最後に，8 章で本研究のまとめと今後の課題について述べる．

2　準備

本章では本研究の前提となる技術および用語について説明する．

2.1　ソースコード理解

ソフトウェアライフサイクルにおいて，作業量全体の 7 割以上を保守作業が占めていると言われている [9]．ソフトウェアの保守とは，ソフトウェアシステムの運用後に行う不具合の除去，パフォーマンスや機能改善のための修正作業である [10]．ソフトウェアの保守を行うには，対象となるソフトウェアのソースコードやドキュメントを元に，ソフトウェアの振る舞いを理解する必要がある．ソフトウェア保守に要する時間的コストの大部分が，ソースコードを読んで理解することであり，ソースコードの理解性・可読性はソフトウェアの保守性に大きな影響を与える [4, 5, 6]．

2.1.1　ソースコード理解性の計測

ソースコードの理解性を表す厳密な基準は存在しない．そのため，理解性を計測するために様々な手段が用いられている．代表的なものは以下の通りである [11, 12]．

擬似的な保守作業

被験者に対し，ソースコードへの具体的な機能追加，デバッグなどのタスクを与え，その正確さや作業に要した時間などからソースコードの理解性を測定する．

ソースコードの再現

被験者に一定時間ソースコードを見せて記憶させ，その後可能な限り再現させたり，必要だと感じれば修正を加えさせたりすることで，元のソースコードの理解性を測定する．

主観的評価

アンケートなどによって，被験者がソースコードをどの程度理解しているかという評価を得る．

2.1.2　ソースコード理解支援

ソフトウェア保守においてソースコード理解に要する時間的コストの割合が高いことから，理解性を向上させることでソフトウェアの保守性も改善することができると考えられる．ソースコードを理解しやすくするための要素や手段として，以下が挙げられる．

コメントの記述

ソースコード中のコメントやドキュメントは，ソースコードの理解を支援する働きがある [13, 14]．Java においては，クラスやメソッド，フィールドなどの要素に対して説明を記述できる，ドキュメンテーションコメントと呼ばれるコメントが存在する．ドキュメンテーションコメントをソースコード中に記述することで，HTML 形式のドキュメントを生成することができる．また，ソースコード中の識別子名などから自動的にメソッドの処理内容を要約する手法が提案されている [15]．

可視化

ソースコードを理解しようとする際，視覚的な情報を利用することは非常に有効な手段である．例えば，テキストエディタ上でプログラミング言語の予約語を強調することで理解性を高めることができる．この機能は多くのエディタで標準的に備わっている．また，Eclipse などの統合開発環境では，予約語の強調表現以外にも理解性を高めるための様々な機能が備わっている [16]．例えば，ある識別子に着目すると，その識別子が出現する全ての箇所がエディタ上でハイライトされ，データフローの追跡が容易となる．また，クラスの継承関係や呼び出し関係を表示することで，ファイル間の移動を短縮することができる．

コーディング規約

ソフトウェア開発・保守は複数人で行うことが一般的であるため，他人の記述したソースコードについても素早く理解できるよう，Java Code Convensions といった開発者が守るべきコーディング規約が存在する [17]．例えば，変数やメソッドなどの識別子は使用意図が分かるような命名を行うべきだと言われている．また，制御構造などの階層構造を認識しやすくするため，ソースコード中のインデント（字下げ）の方法についても言及されている．適切なインデントや空行の存在は理解性の向上に繋がることが確認されている [18]．

2.2 リファクタリング

ソフトウェアの保守において，デバッグや機能改善などソースコードに変更を重ねることでプログラムの設計が劣化することがある [19]．設計が劣化すると理解性や再利用性も悪化する．これを改善するためには，プログラムの振る舞いを変えずに内部構造を改善するリファクタリングという技術が有効である [1]．

2.2.1 リファクタリングプロセス

リファクタリングプロセスは，次の一連の作業から成る [1]．以下のプロセスはそれぞれ異なるツールや技術によって支援されている．

1. リファクタリングすべき箇所を特定する
2. 特定した箇所に対して，どのリファクタリングを適用するか検討する
3. リファクタリング適用後にプログラムの振る舞いが変わらないことを確認する
4. リファクタリングを適用する
5. リファクタリングの効果を測定する
6. リファクタリングが適用されたソースコードと，ドキュメントなど他の成果物との一貫性を保持する

2.2.2 リファクタリングすべき箇所の特定

リファクタリングすべき箇所を特定するには，ソースコードから不吉なにおい（Bad Smell）を検出する方法が一般的である [1]．不吉なにおいとは，ソースコード中に存在する，将来的に問題を引き起こす可能性のある箇所を指し，保守性を低下させる原因であると言われている．そのような不吉なにおいに対する具体的な対処方法は，Fowler によってリファクタリングパターンとしてまとめられている [20]．ソースコードから不吉なにおいを検出し，リファクタリング箇所を特定するツールや手法がこれまでに多く提案されている [21, 22, 23]．代表的な不吉なにおいと，その検出方法は以下の通りである．

重複したコード

ソースコード中で内容が一致または類似したコード片を重複コードと呼ぶ．重複コードの一部に修正が必要な場合，重複関係にあるコードに対しても同様の修正が必要になる可能性があることから，ソフトウェアの保守に影響を及ぼすと考えられている．重複コードに厳密な定義はなく，テキストベースや構文ベースなど様々な基準から重複コードを検出するツールが存在する [24, 25]．

長すぎるメソッド

オブジェクト指向デザインにおいて，メソッドは 1 つの機能的なまとまりを持つべきであると言われ，長すぎるメソッドは理解性や再利用性が低いと考えられている．長すぎるメソッドを検出するには，コード行数などのソフトウェアメトリクスを計測する方法が一般的である．

巨大なクラス

クラスもまた 1 つの役割を担うべきである．多くの役割を持ったクラスはインスタンス変数が増加し重複コードを生み出す原因にもなる．巨大なクラスを特定するには，クラス内のインスタンス変数の数から判断する他，クラス内の変数およびメソッドがどの程度関連しているかを示す凝集度と呼ばれるソフトウェアメトリクスを用いた方法が取られている．

2.2.3 リファクタリングパターン

Fowler が提案したリファクタリングパターンのうち，メソッドの構成に関わる代表的なものを以下に示す．

識別子名の変更

フィールドやメソッドなどの識別子名は，その役割を明確に示すような命名を行うべきである．従って，識別子名の変更は最も簡単で，かつ重要なリファクタリングであると言える．Eclipse にはリファクタリングを支援する機能がいくつか備わっているが，中でも最も良く用いられているものは識別子名の変更であることが示されている [26]．

メソッド抽出

メソッド抽出リファクタリングとは，メソッドの一部を新たなメソッドとして切り出すことである．重複したコードや長すぎるメソッドなど，いくつかの不吉なにおいへの対処法として用いられている．また，他のリファクタリングの前に行われることが多く，最も基本的なリファクタリングの 1 つである [26]．

条件記述の分解

条件分岐の分岐先を新たなメソッドとして抽出することを，特に「条件記述の分解」と呼ぶ．分岐先の処理を 1 つのメソッドとしてまとめ，かつ適切な命名を行うことで，条件記述の意図を明確にすることができる．条件分岐やループの多用はプログラム制御の流れを複雑にし，理解性を低下させる大きな要因である．従って，条件記述の分解はそのような理解性の低下を防ぐことにも繋がる．

2.2.4 リファクタリングの自動化

大規模ソフトウェアから手動でリファクタリングすべき箇所を特定するには非常に手間がかかり，また特定した箇所をどのようにリファクタリングすべきか決定するのにも多くの経験や知識を必要とす

る．このように，リファクタリングを手動で行うことは難しいため，リファクタリング作業を支援する手法が必要である．Eclipse などの統合開発環境では，半自動なリファクタリング機能が備わっていることが多い．半自動とは，リファクタリング箇所の選択や，実際に適用するかどうかの決定など，リファクタリングプロセスの一部を開発者自身が行うことを意味する．半自動なリファクタリングは，特に大規模なソフトウェアに適用する際，開発者の意思決定に時間的コストがかかると報告されている [27]．一方，全自動でリファクタリングを行うツールも存在するが，全自動なリファクタリングは開発者の望まない変更を行う可能性があるなど，開発者にとってのソースコードの理解性を低下させることさえある．最近の研究では，自動リファクタリングツールの多くは用いられていないことが分かっている [26]．この原因として，ツールが広く知れ渡っていないこと，コストパフォーマンスの低さ，複雑な仕様，開発者がリファクタリング効果を予測できないことなどが挙げられている [28]．

2.3 ソフトウェアメトリクス

ソフトウェア保守を行うにあたって，メトリクスを用いたソフトウェアの計測は重要な技術である．メトリクスを計測することで，リファクタリングすべき箇所の特定や，リファクタリングの効果を測定することができる [21, 29, 30]．また，メトリクスを用いて不具合が顕在化しそうなモジュール（フォールトプローンモジュール）を特定し，バグ修正を早期に行うことで，ソフトウェアの品質を改善することも可能である [31, 30]．主にソースコードの複雑さを表現する代表的なメトリクスを以下に示す．

コード行数

ソースコードの行数が大きいということは，規模が大きくそれだけ複雑さも増すと言える．従って，コード行数は基本的な複雑度メトリクスの 1 つとして認識されている．空行やコメントのみの行を除いた行をもってコード行数とすることが多い．

サイクロマチック数

サイクロマチック数とは，McCabe によって提案されたメトリクスである [32]．プログラム制御の流れを有向グラフで表現したときのノードの数を v，エッジの数を e とすると，$e = v + 2$ で表される．この値は直観的にはソースコードの分岐の数に 1 を加えた数である．サイクロマチック数の値が大きいと，テストケースを作成する手間がかかり，保守性が下がると指摘されている．また，McCabe はサイクロマチック数を 10 以下に抑えることが望ましいと述べており，統合開発環境やメトリクス計測ツールでもこの基準が用いられることが多い．

CK メトリクス

CK メトリクスは，オブジェクト指向デザインにおいてクラス構造に基づく複雑度を評価するためのもので，Chidamber と Kemerer によって提案された [33]．クラスの重み・結合度・凝集度・応答数・継承の深さ・子クラスの数の計 6 つのメトリクスが定義されている．CK メトリクスは，特にフォールトプローンモジュールの検出に有効であると言われている [34]．また，クラスの凝集度を計測することでクラス抽出などのリファクタリング候補を特定することができるが，提案されている手法の多くは CK メトリクスに含まれる凝集度や，それを元に新たに提案されたメトリクスを用いている [35]．

3 関連研究

3.1 ソースコード理解性について

Buse と Weimer は様々なメトリクスとソースコード可読性との相関を調査した結果，識別子の数がソースコード可読性に最も影響を与えていることを示している [36]．また，人はソースコードを理解しようとする際，ソースコードを読みながら頭の中で実行することがある．このような作業をメンタルシミュレーションと呼ぶ [12]．Nakamura らは，人の記憶形式をキューでモデル化し，メンタルシミュレーションのコストを計測した [7]．この結果，キューにない変数を参照するとき，すなわち宣言されてから参照されるまでの間に多くの処理を含む変数を参照するとき，コストの増大に繋がるということが分かっている．

ソースコードを理解するための時間のうち，1 つのドキュメントに対してスクロール操作などで移動を行う時間は約 7 分の 1 を占めるという報告がある [6]．Biegel らは，Java ソースコード中のフィールドおよびメソッドの並び方は理解性に影響を与える要因であると考え，その並び方にどのような基準があるか，16 のオープンソース・ソフトウェアを対象に調査を行った [37]．その結果，最も広く用いられている基準は Java Code Convensions で定められている基準であるが，それに続く基準は様々なものが存在することが分かっている．

3.2 プログラム文並べ替えによる理解性向上のための手法

佐々木らは，プログラム文を並べ替えることで変数の宣言と参照との間の距離を短くする手法を提案した [8]．佐々木らの提案手法では，一定の条件を満たすすべての並べ替えパターンを生成した後，その中から変数の宣言・参照間距離が最小のものを出力するという戦略を取っている．こうすることで，変数の宣言・参照間距離が最小となる並びは得られるが，その反面実行時間は長くなってしまう．なぜなら，リファクタリングを行うプログラムの総行数を n としたときに，すべてのプログラム文の並べ替えの組み合わせを計算する時間計算量は $O(n!)$ となってしまい，n が大きくなった場合に現実的な時間で最適解を求めるのは困難なためである．また，佐々木らは提案手法のプロトタイプツールを実装しているが，このツールはすべての機能を実装できておらず，振る舞いが破壊されたプログラムを出力する可能性がある．

3.3 サイクロマチック数とソフトウェア保守性

サイクロマチック数は伝統的に用いられる複雑度メトリクスであるが，他のメトリクスと比べてソフトウェアの保守性を計測するには不向きであることが様々な調査結果によって示されている．例えば，フォールトプローンモジュールの特定にはサイクロマチック数よりも CK メトリクスの方が有用であ

```
444 long lineCount = this.main.compilerStats[0].lineCount;
    ...
448   for (...) {
        ...
455   }
456 long parseTime = parseSum/(length - 2);
    ...
464 String.valueOf(lineCount),
    ...
468 if (...) {
      ...
472   String.valueOf(parseTime),
      ...
481 }
```

図 1　宣言と参照が離れている例

ることが報告されている [34]．また，Buse と Weimer によるソースコードの可読性とメトリクスとの相関についての調査では，サイクロマチック数とソースコードの可読性との相関は低いという結果が得られている [36]．近年ではソフトウェアリポジトリマイニングが盛んに行われ，サイクロマチック数などの複雑度メトリクスよりも，開発履歴などによる履歴メトリクスの方がフォールトプローンモジュールの特定や，ソフトウェアの脆弱性の予測に効果的であることが示されている [38, 39, 40]．

3.4　ソースコード理解性向上のためのリファクタリング手法

エディタ上での強調表現やフォーマットの整形など，ソースコード上の見た目を改善する技術を prettyprinting と呼ぶ [41]．prettyprinting は古くから用いられている技術で，プログラミング言語に依存しない手法は Oppen によって最初に提案された [42]．また，prettyprinting は基本的にプログラムの構文的な情報を元に行われているが，Wang らは構文情報の他にデータ依存も考慮した上でソースコードの意味的なまとまりを識別し，空行で分割することで理解性を向上させる手法を提案した [43]．

Atkinson と King は，行数や文の数などのメトリクスを用いて，低コストでリファクタリング候補を自動検出する手法を提案した [29]．Relf は，ソースコードの可読性を高めるために，ソースコード上の情報から適切な識別子名を特定し，提示する手法を提案した [44]．Tsantalis と Chatzigeorgiou は，プログラム中に存在する全ての変数について，データの依存関係を元に関連性のある文のまとまりを特定し，メソッド抽出リファクタリングの候補を提示する手法を提案した [45]．

3.5　研究目的

図 1 に変数の宣言と参照が離れている例を示す．図中では，変数 `lineCount` および変数 `parseTime` の宣言および参照を強調表示している．これらの変数はともに宣言から初めて参照されるまでに 15 行

以上離れてる．また，変数 parseTime は 468 行目から始まる if 文内でしか用いられていないにもかかわらず，if 文の外で宣言されている．これらのように，宣言されてから参照されるまでに多くの処理をはさむ変数は，ソースコードの理解コストを増大させることが分かっている [7].

そこで本研究では，プログラム文を並べ替えることで変数の宣言と参照を近くするようなリファクタリングを行う手法を提案する．提案手法では，入力として与えられたメソッドに対して，リファクタリング候補，すなわちプログラム文を並べ替えたものの生成と，リファクタリング候補の宣言・参照間距離の評価を繰り返し行うことで，宣言・参照間距離を短縮したメソッドを出力する．

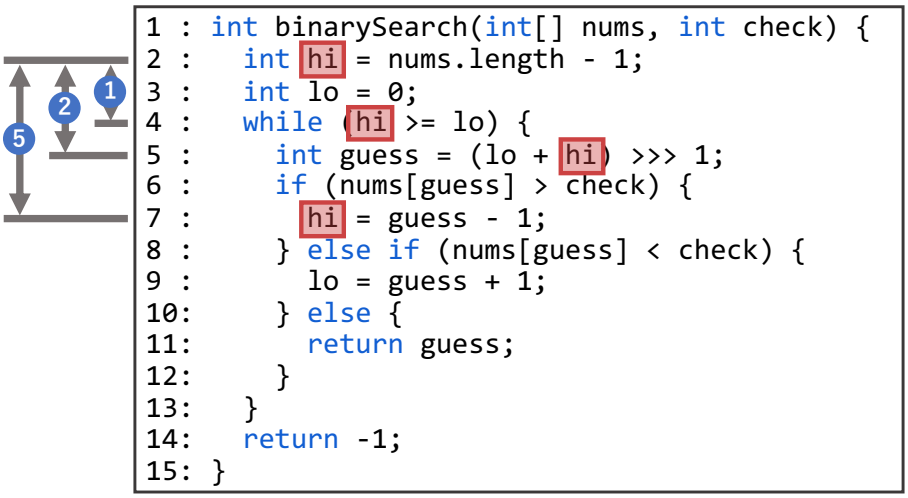

図 2　宣言・参照間距離の例

4　提案手法

本研究では，メソッド内の変数の宣言・参照間距離の総和を小さくするリファクタリングを行う手法を提案する．

4.1　宣言・参照間距離

変数 v の宣言・参照間距離 $distanc(v)$ は，以下の式で定義される．ただし，$Ref(v)$ は変数 v の参照の集合とし，$d(v,r)$ は変数 v の宣言と参照 r の間に存在するプログラム文およびネストの数とする．

$$distance(v) = \sum_{r \in Ref(v)} d(v,r)$$

変数の宣言・参照間の距離の計測方法の例を図 2 に示す．図中に示したメソッド `binarySearch` の 2 行目で変数 `hi` が宣言されている．変数 `hi` の宣言・参照間距離について考える．変数 `hi` は 4 行目，5 行目および 7 行目の 3 箇所で参照されている．これらをそれぞれ，r_4, r_5, r_7 とする．変数 `hi` の宣言と r_4 の間にはプログラム文が 1 つ存在するため，$d(\texttt{hi}, r_4) = 1$ となる．また，`hi` の宣言と r_5 の間にはプログラム文が 1 つおよび，`while` 文のネストが 1 つ存在するため，$d(\texttt{hi}, r_5) = 2$ となる．同様に，$d(\texttt{hi}, r_7) = 5$ となり，$disitance(\texttt{hi}) = 1 + 2 + 5 = 8$ となる．

4.2　提案手法の概要

本研究では，探索的に変数の宣言・参照間距離を短くするようなリファクタリングを行う手法を提案する．提案手法の概要を図 3 に示す．提案手法の入力は，リファクタリングしたいメソッドとテスト集

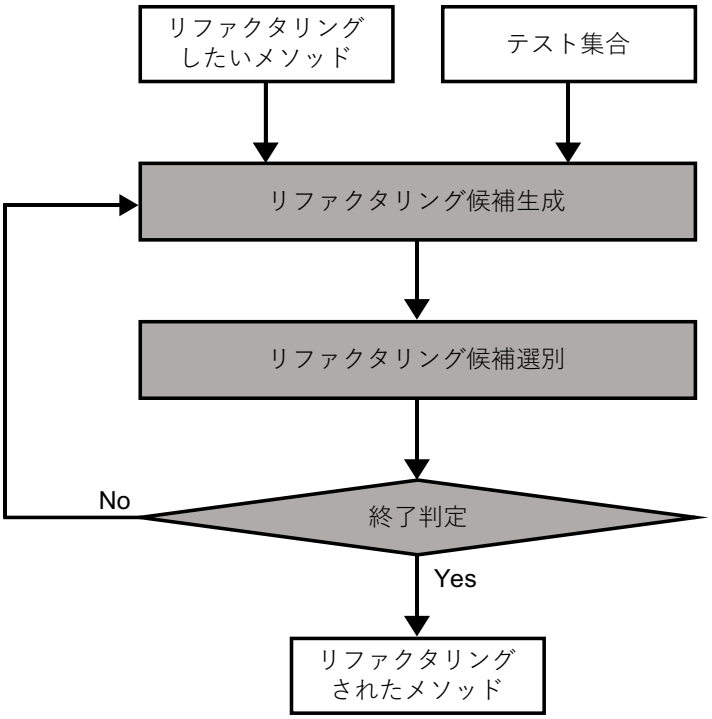

図 3　提案手法の概要

合である．出力は，リファクタリング後のメソッドである．入力として与えるテスト集合は，リファクタリングしたいメソッドの振る舞いを表しており，メソッドの仕様として解釈できる．

提案手法は以下の 3 つのステップで構成される．ステップ 1〜3 は終了条件が成り立つまで繰り返される．このステップ 1〜3 のループのことを以降メインループと呼ぶ．

ステップ 1　リファクタリング候補生成

ステップ 2　リファクタリング候補選別

ステップ 3　終了判定

各ステップについて説明する．

4.2.1　ステップ 1 リファクタリング候補生成

ステップ 1 では，リファクタリング候補の生成を行う．リファクタリング候補とは入力されたメソッドに対して，プログラム文の並べ替えを行ったものである．ステップ 1 で行う処理は主に以下の 3 つである．

- プログラム文並べ替え対象の決定
- プログラム文並べ替え実行
- リファクタリング候補の評価

各処理について説明する．

まず，プログラム文の並べ替えを行う対象，つまり何に対してプログラム文の並べ替えを行うかを決定する．プログラム文並べ替え対象は，メインループの初回実行時と 2 回目以降の実行時で異なる．メインループの初回実行時には入力されたメソッドに対してプログラム文の並べ替えを行い，2 回目以降のメインループの実行時には，それ以前に生成したリファクタリング候補に対して並べ替えを行う．2 回目以降のメインループ実行時にどのリファクタリング候補に対してプログラム文の並べ替えを行うかは，後述する評価値によって重み付けされた乱択によって決定する．評価値が高いリファクタリング候補ほど選択されやすくなる．

次に，プログラム文の並べ替えを行う．1 回の並べ替えでは 1 つのプログラム文の移動を行う．どのプログラム文をどこに移動するかはランダムに決定する．

最後に，生成したリファクタリング候補の評価を行う．評価は 2 つの側面から行う．1 つはメソッド内で参照されている各ローカル変数の宣言・参照間距離である．各ローカル変数の宣言・参照間距離の総和が短いものほど評価値が高くなる．もう 1 つの評価は，テストの通過率である．テスト通過率が高いリファクタリング候補ほど評価値が高くなる．テストは入力されたメソッドの仕様とみなすことができるため，通過しないテストが存在することは，元の振る舞いを破壊していることを意味する．しかし，一旦振る舞いが破壊されたとしても，さらなる変更を加えることで元の振る舞いをするようになる可能性もある．振る舞いが破壊されたリファクタリング候補も評価値を低く設定し，残しておくことでリファクタリング候補に多様性が生まれ局所解に陥ることを防ぐことができると考え，提案手法ではこのような評価値の計算を行う．

4.2.2　ステップ 2 リファクタリング候補選別

ステップ 2 では，ステップ 1 で生成したリファクタリング候補の中から，評価値が高いものを一定数残し，それ以外を破棄する．リファクタリング候補をいくつ残すかは，提案手法の実行時に利用者に入力してもらう．評価値が高いリファクタリング候補を優先的に残し，それらに対してさらなる変更を加えることで，より評価値の高いリファクタリング候補が生成されることを期待する．

4.2.3　ステップ 3 終了判定

ステップ 3 では，リファクタリング候補の探索を終了するかどうかの判定を行う．終了条件は，提案手法の実行時に利用者が入力した制限時間が経過したかどうかである．制限時間を経過していれば，探

索を打ち切り，その時点で見つかったリファクタリング候補のうち，元の振る舞いを保ちつつ，変数の宣言・参照間距離がより短いものを出力する．

5 実装

4 章で述べた提案手法を Java で記述されたプログラムに対して実行するツールを実装した．実装したツールの入力は以下の通り．

- リファクタリングしたいメソッドを含むソフトウェアのソースコード
- リファクタリングしたいメソッドを含むクラスに対するテスト集合

提案手法では，リファクタリング候補のビルドおよびテストを実行する必要があるため，リファクタリングしたいメソッドだけでなく，ソフトウェア全体のソースコードを入力する必要がある．また実行時パラメータとして，どのメソッドをリファクタリングするか，および制限時間を受け取る．

出力は，リファクタリングを適用したソースコードおよび，入力されたソースコードとリファクタリングを適用したソースコードの差分ファイルの 2 つである．

提案手法は，探索的にリファクタリングを行う性質上，ソフトウェアのビルドおよびテスト実行を何度も繰り返し行う必要がある．ソフトウェアによってはビルドおよびテスト実行に 10 分以上かかる場合もあるため，実装したツールでは高速化のために以下の 2 つの工夫を行った．

- ファイル IO のインメモリ化
- 差分ビルド

ファイル IO のインメモリ化

ビルドとテスト実行には多くのファイル IO が発生する．具体的には，ビルド時のソースコードの読み込みや，ビルド結果の書き込み，およびテスト実行対象のバイナリの読み込みなどである．これらにかかる時間を短縮するために，実装したツールではメモリ上に論理的なファイルシステムを構築し，ソースコードやバイナリをこのメモリファイルシステム上に配置する．これにより，ツール実行の最初期に行われるソースコードの読み込み，および終了時に行われる結果の出力以外のファイル IO をメモリ内で完結することができる．

差分ビルド

テストを実行するには，リファクタリングしたいメソッドが他のクラスに対する依存を持っている可能性もあるため，ソフトウェア全体のビルド結果が必要である．しかし，提案手法実行中にはリファクタリングしたいメソッド以外に対する変更は行われないため，その他のクラスに対する再ビルドを行う必要はない．そこで実装したツールでは，変更が加えられていない部分のビルド結果をキャッシュして

おき，不必要なビルドを回避する工夫を行っている．

6　評価実験

提案手法を用いてプログラムの理解性を向上させられるかを確かめるために，以下の 2 つの実験を行った．

実験 1.　リファクタリング結果の質の評価

実験 2.　開発者が記述した可読性の高いプログラムの再現

以降では，それぞれの実験について述べる．

6.1　実験 1. リファクタリング結果の質の評価

提案手法によって出力されたリファクタリングにより，プログラムの理解性が向上するかを評価するために，オープンソースソフトウェアに対して提案手法を適用した結果をそのソフトウェアの開発者に提示し，理解性についてアンケートを実施する実験を行った．この実験の内容と結果について述べる．

6.1.1　実験対象

実験対象は，kGenProg [46] というオープンソースソフトウェアに含まれる 8 個のメソッドである．kGenProg の詳細を表 1 に示す．また，実験を行ったメソッドを表 2 に示す．

6.1.2　実験方法

6.1.1 節で述べた実験対象に対して，オリジナルのメソッドと提案手法によってリファクタリングされたメソッドのどちらが理解しやすいかを被験者に答えてもらうアンケートを行った．本実験の手順を以下に示す．

1. 各メソッドに対して，提案手法を実行する．
2. 提案手法がリファクタリングしたメソッドのインデントや改行位置なのどのフォーマットをオリジナルのものと統一する．

表 1　kGenProg の詳細情報

総行数 (LoC)	18,828
テストケース数	269
テストの命令網羅率	78.76%
実験を行ったリビジョン（日付）	5242340fc9 (2019/1/31)

3. 被験者に対して，リファクタリング前後のメソッドを提示し，理解のしやすさについて，以下の選択肢から選んでもらう．
   - リファクタリング前のメソッドの方が理解しやすい
   - リファクタリング後のメソッドの方が理解しやすい
   - 理解のしやすさに違いはない

この実験の被験者は実験対象ソフトウェアである kGenProg の開発者 7 名である．

6.1.3 実験結果

実験結果を図 4 に示す．グラフ中の縦棒は各メソッドに対する被験者の回答の内訳を示しており，青色が「リファクタリング後のメソッドの方が理解しやすい」，灰色が「リファクタリング前後で理解のしやすさに違いはない」，赤色が「リファクタリング後のメソッドの方が理解しやすい」をそれぞれ表している．

グラフから，すべてのメソッドに関して，リファクタリング後のメソッドの方が理解しやすいと回答した人数が，リファクタリング前のメソッドの方が理解しやすいと回答した人数を上回っている．特に，メソッド K4 に関しては全員がリファクタリング後のメソッドの方が理解しやすいと回答している．

以上のことから，提案手法によるリファクタリングにより，理解性が向上したと考えることができる．

表 2　実験対象のメソッド

メソッド ID	メソッド名
K1	jp.kusumotolab.kgenprog.ga.mutation.RandomMutation#exec
K2	jp.kusumotolab.kgenprog.output.PatchGenerator#exec
K3	jp.kusumotolab.kgenprog.output.PatchGenerator#makeFileDiff
K4	jp.kusumotolab.kgenprog.output.TestResultsSerializer#serialize
K5	jp.kusumotolab.kgenprog.output.VariantStoreExporter#writeToFile
K6	jp.kusumotolab.kgenprog.project.factory.EclipseProjectFactory#create
K7	jp.kusumotolab.kgenprog.project.jdt.ASTStream#next
K8	jp.kusumotolab.kgenprog.project.jdt.JDTASTConstruction#constructAST

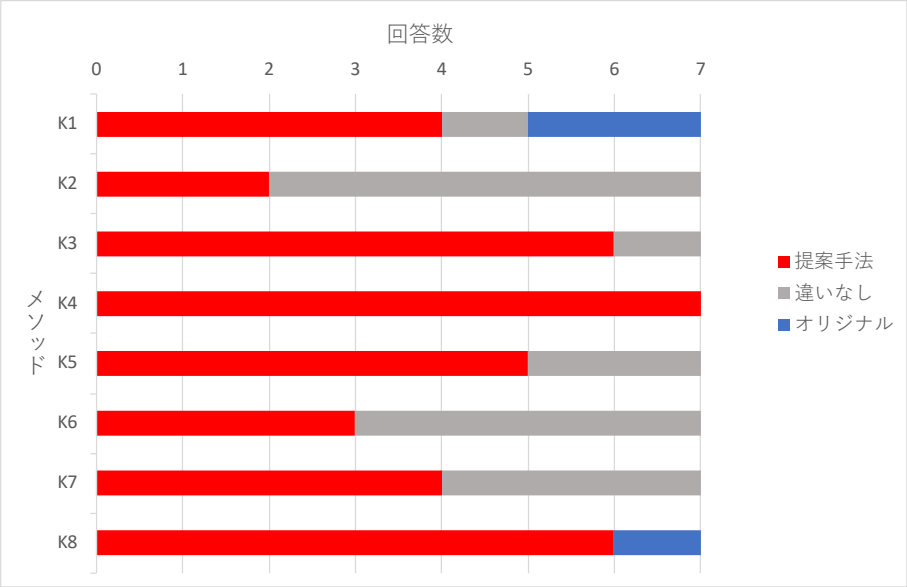

図 4　各メソッドに対する回答の内訳

6.1.4　考察

図 4 より，メソッド K1 はリファクタリング前のメソッドの方が理解しやすいと回答した人数が最も多かった．提案手法がメソッド K1 に対して行ったリファクタリングの詳細を図 5 に示す．このリファクタリングでは，3 変数 `generatedVariants`, `weightFunction`, `gene` の移動が行われている．

このリファクタリングに対してリファクタリング前の方が理解しやすいと回答した 2 人の開発者にその理由を尋ねた．その結果，両者からとも `weightFunction` の移動は理解性の向上につながるとの回答が得られた．一方，`generatedVariants`, `gene` の移動は理解性に変化はない，もしくは理解性が減少したとの回答が得られた．`generatedVariants` の移動が理解性の減少につながった理由としては，以下のような回答が得られた．

- メソッドの最後で `return` する変数はメソッドの冒頭で宣言している方が理解しやすい．
- `generatedVariants` や `currentVariants` といった型や名前が類似する変数はまとめて宣言していた方が理解しやすい．

また，`gene` の移動が理解性の減少につながった理由としては，`gene` は `makeGene` メソッドの返り値によって初期化されており，これは `makeBase` メソッドの返り値によって初期化されている `base` の変数宣言とまとめた方が理解しやすいという回答が得られた．

提案手法による変数の移動が理解性の減少につながった理由として，類似構造を持つ変数の宣言はま

```
--- jp.kusumotolab.kgenprog.ga.mutation.RandomMutation
+++ jp.kusumotolab.kgenprog.ga.mutation.RandomMutation
@@ -36,8 +36,6 @@
   @Override
   public List<Variant> exec(final VariantStore variantStore) {

-    final List<Variant> generatedVariants = new ArrayList<>();
-
     final List<Variant> currentVariants = variantStore.getCurrentVariants();

     final Roulette<Variant> variantRoulette = new Roulette<>(currentVariants, e ->
         ↪ {
@@ -45,22 +43,20 @@
       final double value = fitness.getValue();
       return Double.isNaN(value) ? 0 : value + 1;
     }, random);
+    final List<Variant> generatedVariants=new ArrayList<>();

     for (int i = 0; i < mutationGeneratingCount; i++) {
       final Variant variant = variantRoulette.exec();
       final List<Suspiciousness> suspiciousnesses = variant.getSuspiciousnesses();
-      final Function<Suspiciousness, Double> weightFunction = susp -> Math.pow(
    ↪ susp.getValue(), 2);
-
       if (suspiciousnesses.isEmpty()) {
         continue;
       }

+      final Function<Suspiciousness, Double> weightFunction=susp->Math.pow(susp.
    ↪ getValue(),2);
       final Roulette<Suspiciousness> roulette =
           new Roulette<>(suspiciousnesses, weightFunction, random);
       final Suspiciousness suspiciousness = roulette.exec();
       final Base base = makeBase(suspiciousness);
-      final Gene gene = makeGene(variant.getGene(), base);
       final HistoricalElement element = new MutationHistoricalElement(variant,
           ↪ base);
+      final Gene gene = makeGene(variant.getGene(), base);
       generatedVariants.add(variantStore.createVariant(gene, element));
     }

   return generatedVariants;
  }
```

図 5　提案手法が K1 に対して行ったリファクタリング

とめた方が理解性の向上につながるという意見は `generatedVariants` と `gene` の両方に共通する．この結果は，佐々木らの報告と合致する [8].

以上のことより，提案手法に対して以下の機能を追加することで，理解性をより向上させられる可能性がある．ただし，これらの機能によって理解性が向上するかは開発者によって個人差があると考えられるため，これらの機能を利用するかどうかは開発者が選択できるように実装すべきであると考えられる．

- `return` 文中で参照されている変数を並べ替えない．
- 同一の型や類似した名前などを持つ変数が連続して宣言されている場合は並べ替えない．

6.2 実験 2. 開発者が記述した可読性の高いプログラムの再現

提案手法を用いて，開発者が記述した可読性の高いプログラムを再現できるかを確認するために評価実験を行った．この実験の内容と結果について述べる．

6.2.1 実験対象

実験対象は，オープンソースソフトウェアの Apache Commons Math [47] に含まれる 6 個のメソッドである．Apache Commons Math の詳細を表 3 に示す．また，実験を行ったメソッドを表 4 に示す．

6.2.2 実験方法

6.2.1 節で述べた実験対象に対して，提案手法を用いて開発者が記述した可読性の高いプログラムを再現できるかを確かめるための実験を行った．実験手順を以下に示す．

1. 各メソッドからコメントを取り除く．
2. メソッド内で宣言されている各変数を，振る舞いを変化させない範囲で，できるだけ先頭側に移動させる．
3. 各メソッドに対して，提案手法を実行する．

表 3　Apache Commons Math の詳細情報

総行数 (LoC)	307,182
テストケース数	4,890
テストの命令網羅率	90.17%
実験を行ったリビジョン（日付）	d7d4e4df72 (2018/12/11)

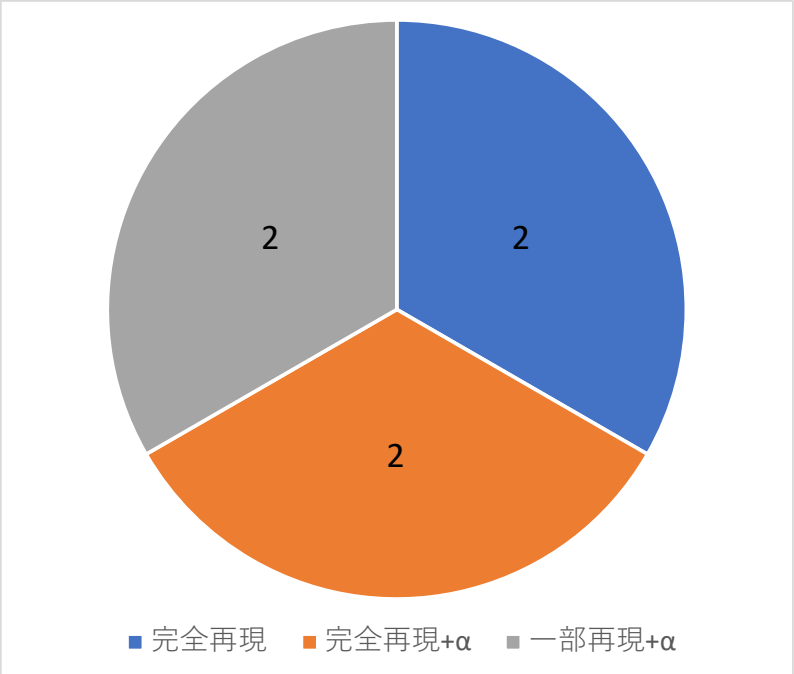

図 6　実験結果

4. 提案手法がリファクタリングしたメソッドのインデントや改行位置などのフォーマットをオリジナルのものと統一する．
5. 2. で変数を移動させたメソッドと提案手法によるリファクタリングが行われたメソッドを比較し，開発者の記述を再現できているかを確かめる．

表 4　実験対象のメソッド

メソッド ID	メソッド名
M1	`org.apache.commons.math4.util.KthSelector#partition`
M2	`org.apache.commons.math4.stat.interval.ClopperPearsonInterval#createInterval`
M3	`org.apache.commons.math4.stat.interval.NormalApproximationInterval#createInterval`
M4	`org.apache.commons.math4.transform.FastCosineTransformer#fct`
M5	`org.apache.commons.math4.transform.FastCosineTransformer#transform`
M6	`org.apache.commons.math4.transform.FastSineTransformer#fst`

6.2.3 実験結果

実験結果を確認したところ，開発者の記述をまったく再現できていないものは存在しなかった．つまり，どのメソッドに対しても提案手法は開発者の記述を少なくとも 1 つは再現できていた．変数を移動させたメソッドと提案手法によるリファクタリングが行われたメソッドを比較し，次の 3 つに分類した．

- 開発者の記述を完全に再現できた．
- 開発者の記述を完全に再現した上で，さらなる変更が加えられた．
- 開発者の記述を一部を再現した上で，さらなる変更が加えられた．

この結果を図 6 に示す．図中の青色が「開発者の記述を完全に再現できたメソッドの数」，オレンジ色が「開発者の記述を完全に再現した上で，さらなる変更が加えられたメソッドの数」，灰色が「開発者の記述を一部再現した上で，さらなる変更が加えられたメソッドの数」をそれぞれ表している．図から分かる通り，各分類に 2 つずつのメソッドが存在した．

6.2.4 考察

提案手法はメソッド M5 の開発者の記述を完全に再現できた．提案手法がメソッド M5 に対して行ったリファクタリングの詳細を図 7 に示す．メソッド M5 に対しては，提案手法を実行する前に，3 変数 `s`, `s2`, `s1` をメソッドの先頭に移動した．提案手法を実行した結果，これら 3 変数は開発者が記述した元の位置に移動された．

提案手法はメソッド M4 の開発者の記述を完全に再現した上で，さらなる変更を加えた．提案手法がメソッド M4 に対して行ったリファクタリングの詳細を図 8 に示す．メソッド M4 に対しては，提案手法を実行する前に，2 変数 `x`, `t1` をメソッド冒頭部分に移動した．提案手法を実行した結果，これら 2 変数は開発者が記述した元の位置に移動された．

これに加えて，提案手法は 3 変数 `transformed`, `a`, `c` の移動も行った．変数 `transformed` はメソッドの先頭で宣言されていたものが，提案手法により参照に最も近い位置に移動されている．これにより宣言・参照間距離は減少している．しかし，`transformed` はメソッドの最後で `return` される変数である．6.1.4 節の開発者の意見でも述べた通り，メソッドの最後で `return` される変数をメソッドの先頭で宣言した方が理解しやすいという開発者もいるため，この変更は一概に可読性を向上させるものとは言えない．

変数 `a` および `c` は類似した名前を持つ変数 `b` と辞書順に並べて宣言されていたが，提案手法により参照の直前に移動された．これにより宣言・参照間距離は減少したが，6.1.4 節の開発者の意見でも述

```
--- org.apache.commons.math4.transform.FastCosineTransformer
+++ org.apache.commons.math4.transform.FastCosineTransformer
@@ -90,17 +90,16 @@
     @Override
     public double[] transform(final double[] f, final TransformType type)
       throws MathIllegalArgumentException {
-        final double s = FastMath.sqrt(2.0 / (f.length - 1));
-        final double s2 = 2.0 / (f.length - 1);
-        final double s1;
-
         if (type == TransformType.FORWARD) {
             if (normalization == DctNormalization.ORTHOGONAL_DCT_I) {
+                final double s = FastMath.sqrt(2.0 / (f.length - 1));
                 return TransformUtils.scaleArray(fct(f), s);
             }
             return fct(f);
         }
+        final double s2 = 2.0 / (f.length - 1);
+        final double s1;
         if (normalization == DctNormalization.ORTHOGONAL_DCT_I) {
             s1 = FastMath.sqrt(s2);
         } else {
             s1 = s2;
```

図 7　提案手法が M5 に対して行ったリファクタリング

べた通り，類似する構造を持つ変数はまとめて宣言した方が理解しやすいという開発者もいるため，この変更は一概に可読性を向上させるものとは言えない．

```
--- org.apache.commons.math4.transform.FastCosineTransformer
+++ org.apache.commons.math4.transform.FastCosineTransformer
@@ -138,34 +138,32 @@
     protected double[] fct(double[] f)
         throws MathIllegalArgumentException {

-        final double[] transformed = new double[f.length];
-
         final int n = f.length - 1;

-        final double[] x = new double[n];
-        double t1 = 0.5 * (f[0] - f[n]);
-
         if (!ArithmeticUtils.isPowerOfTwo(n)) {
             throw new MathIllegalArgumentException(
                 LocalizedFormats.NOT_POWER_OF_TWO_PLUS_ONE,
                 Integer.valueOf(f.length));
         }
+        final double[] transformed = new double[f.length];
         if (n == 1) {       // trivial case
             transformed[0] = 0.5 * (f[0] + f[1]);
             transformed[1] = 0.5 * (f[0] - f[1]);
             return transformed;
         }

+        final double[] x = new double[n];
         x[0] = 0.5 * (f[0] + f[n]);
         x[n >> 1] = f[n >> 1];
+        double t1 = 0.5 * (f[0] - f[n]);
         for (int i = 1; i < (n >> 1); i++) {
-            final double a = 0.5 * (f[i] + f[n - i]);
             final double b = FastMath.sin(i * FastMath.PI / n) * (f[i] - f[n - i])
                 ↪ ;
-            final double c = FastMath.cos(i * FastMath.PI / n) * (f[i] - f[n - i])
    ↪ ;
+            final double a = 0.5 * (f[i] + f[n - i]);
             x[i] = a - b;
             x[n - i] = a + b;
+            final double c = FastMath.cos(i * FastMath.PI / n) * (f[i] - f[n - i])
    ↪ ;
             t1 += c;
         }
         FastFourierTransformer transformer;
```

図 8　提案手法が M4 に対して行ったリファクタリング

7　妥当性への脅威

実験 1 の被験者はいずれもソフトウェア工学に携わっているため，被験者のソースコード読解力に偏りがあると考えられる．そのため，Java のプログラミング経験が少ない者を被験者に含めるなど，多様な被験者を対象とした場合，実験結果が異なる可能性がある．

また，実験 1 および実験 2 ともに対象のソフトウェアは 1 つで，提案手法を適用したメソッドの数も 10 個に満たない．したがって，その他のソフトウェア，多くのメソッドに対して提案手法を適用した場合，異なる結果が得られる可能性がある．

8　おわりに

本研究では，変数の宣言・参照間距離を短縮するようなプログラム文の並べ替えを探索的に行う手法を提案した．提案手法では，入力として与えられたメソッドに対して，リファクタリング候補，すなわちプログラム文を並べ替えたものの生成と，リファクタリング候補の宣言・参照間距離の評価を繰り返し行うことで，宣言・参照間距離を短縮したメソッドを出力する．

オープンソースソフトウェアに含まれるメソッド 8 個に対して提案手法を適用し，提案手法の適用前後のソースコードのどちらが理解性が高いかを回答してもらうアンケートを，実験対象ソフトウェアの開発者 7 名に対して実施した．その結果，すべての実験対象に対して，提案手法を適用後のソースコードの方が理解しやすいと回答した人数が提案手法適用前の方が理解しやすいと回答した人数を上回った．

また，開発者が記述した可読性が高いプログラムに含まれる変数の宣言・参照間距離が長くなるように改変したプログラムに対して，提案手法を適用することで元の開発者の記述を再現できるかを確かめるための実験を行った．その結果，どの実験対象に対しても，少なくとも 1 つの変数は開発者の記述を再現できることを示した．

さらに，実験結果の考察を通して，`return` 文内で参照されている変数はメソッドの冒頭部分で宣言している方が理解しやすく感じる開発者もいるという知見を得た．

今後の課題としては，`return` 文内で参照されている変数や，連続して宣言された類似した構造を持つ変数など，移動させることでかえって理解性が減少してしまう変数の移動を行わないようにする機能の実装が挙げられる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，暖かく励ましていただきました 楠本 真二 教授に心より感謝申し上げます．

本研究の全過程において，終始熱心かつ丁寧なご指導を頂き，多大なるご助力，ご協力を頂きました肥後 芳樹 准教授に心より感謝申し上げます．

本研究に関して，的確で丁寧なご助言を頂き，また日々の研究室生活を盛り上げて頂きました，柗本 真佑 助教に心より感謝申し上げます．

本研究を進めるにあたり，様々な形で励まし，協力を頂いたその他の楠本研究室の皆様に深く感謝申し上げます．

最後に，本研究に至るまでに，講義やセミナー等でお世話になりました大阪大学大学院情報科学研究科の諸先生方に，この場を借りて心より御礼申し上げます．

参考文献

[1] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, pp. 126–139, 2004.

[2] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, p. 50, 2012.

[3] Thainá Mariani and Silvia Regina Vergilio. A systematic review on search-based refactoring. *Information and Software Technology*, Vol. 83, pp. 14–34, 2017.

[4] Adele Goldberg. Programmer as reader. *IEEE Software*, Vol. 4, No. 5, pp. 62–70, September 1987.

[5] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Cubranic. The emergent structure of development tasks. In *Proceedings of th 19th European Conference on Object-Oriented Programming*, pp. 33–48, 2005.

[6] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, Vol. 32, No. 12, pp. 971–987, December 2006.

[7] Masahide Nakamura, Akito Monden, Tomoaki Itoh, Ken-ichi Matsumoto, Yuichiro Kanzaki, and Hirotsugu Satoh. Queue-based cost evaluation of mental simulation process in program comprehension. In *Proceedings of 9th IEEE International Software Metrics Symposium*, pp. 351–360, September 2003.

[8] 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, No. 2, pp. 939–946, 2 2014.

[9] Barry Boehm and Victor R Basili. Software defect reduction top 10 list. *Foundations of Empirical Software Engineering: The Legacy of Victor R. Basili*, Vol. 426, No. 37, 2005.

[10] Salvatore Mamone. The ieee standard for software maintenance. *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 1, pp. 75–76, 1994.

[11] Elliot Soloway and Kate Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, No. 5, pp. 595–609, 1984.

[12] Alastair Dunsmore and Marc Roper. A comparative evaluation of program comprehension measures. *The Journal of Systems and Software*, Vol. 52, No. 3, pp. 121–129, 2000.

[13] Ninus Khamis, René Witte, and Juergen Rilling. Automatic quality assessment of source code comments: the javadocminer. In *Proceedings of the 15th International Conference on Application of Natural Language to Information Systems*, pp. 68–79, 2010.

[14] Eriko Nurvitadhi, Wing Wah Leung, and Curtis Cook. Do class comments aid java program understanding? In *Frontiers in Education, 2003. FIE 2003 33rd Annual*, Vol. 1, pp. T3C–T3C, 2003.

[15] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd International Conference on Software Engineering - Volume 2*, pp. 223–226, 2010.

[16] Eclipse. `https://www.eclipse.org/`.

[17] Java code convensions. `https://www.oracle.com/technetwork/java/codeconvtoc-136057.html`.

[18] Richard J Miara, Joyce A Musselman, Juan A Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, Vol. 26, No. 11, pp. 861–867, 1983.

[19] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, Vol. 27, No. 1, pp. 1–12, 2001.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[21] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pp. 30–38, March 2001.

[22] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 20, No. 6, pp. 435–461, 2008.

[23] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pp. 97–106, 2002.

[24] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.

[25] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable

and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pp. 96–105, 2007.

[26] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, pp. 287–297, 2009.

[27] Frank W Calliss. Problems with automatic restructurers. *ACM SIGPLAN Notices*, Vol. 23, No. 3, pp. 13–21, 1988.

[28] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 233–243, June 2012.

[29] Darren C. Atkinson and Todd King. Lightweight detection of program refactorings. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pp. 663–670, 2005.

[30] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of 34th International Conference on Software Engineering*, pp. 200–210, June 2012.

[31] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of 28th International Conference on Software Engineering*, pp. 452–461, May 2006.

[32] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, pp. 308–320, December 1976.

[33] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476–493, June 1994.

[34] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, pp. 751–761, October 1996.

[35] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, Vol. 84, No. 3, pp. 397–414, 2011.

[36] Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, Vol. 36, No. 4, pp. 546–558, July 2010.

[37] Benjamin Biegel, Fabian Beck, Willi Hornig, and Stephan Diehl. The order of things: How developers sort fields and methods. In *Proceedings of 26th IEEE International Conference on*

Software Maintenance, pp. 88–97, September 2012.

[38] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, Vol. 37, No. 6, pp. 772–787, November 2011.

[39] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of 30th International Conference on Software Engineering*, pp. 181–190, May 2008.

[40] Yasutaka Kamei, Shinsuke Matsumoto, Akito Monden, Ken-ichi Matsumoto, Bram Adams, and Ahmed E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of 26th IEEE International Conference on Software Maintenance*, pp. 1–10, September 2010.

[41] Stoney Jackson, Premkumar T. Devanbu, and Kwan-Liu Ma. Stable, flexible, peephole pretty-printing. *Science of Computer Programming*, Vol. 72, No. 1-2, pp. 40–51, 2008.

[42] Dereck C Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, pp. 465–483, 1980.

[43] Xiaoran Wang, Lori Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *Proceedings of the 18th Working Conference on Reverse Engineering*, pp. 35–44, October 2011.

[44] Phillip A Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *2005 International Symposium on Empirical Software Engineering*, pp. 10–pp, 2005.

[45] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pp. 119–128, 2009.

[46] Yoshiki Higo, Shinsuke Matsumoto, Ryo Arima, Akito Tanikado, Keigo Naitou, Junnosuke Matsumoto, Yuya Tomida, and Shinji Kusumoto. kgenprog: A high-performance, high-extensibility and high-portability apr system. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*, pp. 697–698, 12 2018.

[47] `http://commons.apache.org/proper/commons-math/`.