修士学位論文

題目

プログラム構造に着目したソースコード理解性向上のための リファクタリング支援手法

指導教員

楠本　真二　教授

報告者

佐々木 唯

平成 25 年 2 月 5 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 24 年度 修士学位論文

プログラム構造に着目したソースコード理解性向上のための
リファクタリング支援手法

佐々木 唯

内容梗概

ソフトウェア保守を行うにあたって，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．そのため，ソースコードの理解性を向上させることで，ソフトウェアの保守作業全体のコストを削減することができる．ソースコードの理解性を向上させる手段の 1 つとして，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存在する．本研究では，ソースコードの理解性を向上させるためのリファクタリング支援として，2 つの手法を提案する．

1 つ目は，リファクタリングすべき箇所を特定するため，理解性の低いモジュールを検出する手法である．一般的な方法として，サイクロマチック数や行数といったメトリクスを計測し，これらの値が大きければ理解性が低いとみなす方法が挙げられる．しかし，連続して記述された if 文など，ソースコード中に類似した構造の繰り返しが存在する場合，メトリクスの値が大きくても理解性が低いとは限らない．そこで，メトリクスを計測する前処理として，繰り返し構造を折りたたんでプログラム構造を簡略化する手法を提案する．提案手法を適用した場合としなかった場合のメトリクスを計測して比較を行ったところ，提案手法を適用して計測したメトリクスの方が，理解性の低いモジュールを特定するのに有用であることが分かった．

2 つ目は，ソースコードの理解性を向上させるための，プログラム文の並び替え手法である．既存研究として，ソースコードを読んで理解しようとする際，変数が定義されてから参照されるまでの距離が離れていると理解するためのコストが増大することが報告されている．従って，文の並びはソースコードの理解性に影響を及ぼすと考えられる．そこで，変数の定義・参照間の距離に着目して，モジュール内の文を並び替える手法を提案する．提案手法を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モジュールについて被験者からの評価を得たところ，並び替えの行われたモジュールは理解性が向上するという結果が得られた．

主な用語

ソースコード理解
リファクタリング
メトリクス計測

ソースコード解析
データフロー解析

目 次

図 目 次

表 目 次

第 1 部

背景

1　はじめに

近年，ソフトウェアの大規模化，複雑化に伴い，ソフトウェア保守に要する作業量が増大している．ソフトウェアライフサイクルにおいて，保守作業量が占める割合は非常に高い [1]．更に，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, 3, 4]．そのため，ソースコードの理解性を向上させることは保守作業全体のコスト削減に繋がると考えられ，ソースコード理解に関する研究がこれまでに多く行われている．

ソースコードの理解性・可読性に厳密な定義は存在しない．Buse と Weimer は，テキストの理解のしやすさを可読性（Readability）と定義した上で，ソースコードの可読性を測定するため，識別子，特定の記号，インデントや空行などのフォーマット，コメントといったソースコード上の様々な特徴との相関について調査を行った [5]．また，ソースコードの可読性を向上させるために開発者が守るべきコーディング規約をまとめたものとして，Java Code Convensions などが存在する [6]．例えば，メソッドや変数は使用意図が分かるよう，長すぎず短すぎない命名を行うべきであると述べられている．また，適切な空行も可読性を向上させるための重要な要因の 1 つであると述べられており，Buse と Weimer の調査結果においても，空行はコメントよりも重要度が高いことが分かっている．

Wang らは，ソースコード中の文から意味のあるまとまりを識別し，その間に空行を挿入することでソースコードの可読性を向上させる手法を提案している [7]．このように，ソースコードの可読性を向上させるための手法やツールは多く存在し，これらはリファクタリング操作として考えることができる．リファクタリングとは，プログラムの振る舞いを変えずに内部構造を変化させる技術である．ソースコードには将来的に問題を引き起こす可能性のある「不吉なにおい」が存在し，保守性を低下させる原因であるといわれている．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 [8]．

本研究では，ソースコードの理解性・可読性を向上させるためのリファクタリング手法を新たに提案する．具体的には，以下の点に着目した研究を行う．

1. 理解性の低いモジュールを特定するための，プログラム構造簡略化

2. ソースコードの可読性を向上させるための，プログラム文並び替え

まず，1. について説明する．リファクタリングプロセスにおける第一歩は，リファクタリング候補を特定することである [9]．しかし，大規模なソフトウェアの中から，リファクタリングすべき箇所を特定することは非常に手間のかかる作業である．そこで，メトリクスを用いてソフトウェアを計測し，品質の低い箇所を特定する方法が考えられる [10, 11, 12]．McCabe のサイクロマチック数 [13] は，モジュールの制御パス数を表すメトリクスで，伝統的に用いられる複雑度メトリクスの 1 つである．最近の研究では，ソースコード可読性とほとんど相関がない，モジュール間の関係を考慮していないなど，ソフトウェア品質を計測するにあたってあまり有用ではないことが述べられている [5, 14, 15, 16]．一方，サイクロマチック数の値が人の感じる複雑さとギャップがある例として，if 文

や case 文などの分岐が連続して記述されている例が挙げられている [17]．そこで本研究では，メトリクスを用いて理解性の低いモジュールを特定する，という状況に焦点を当て，それをより効率的に行うためのメトリクス計測の前処理を提案する．提案手法の有用性を確認するために，約 13,000 のオープンソース・ソフトウェアに対して提案手法適用前後のメトリクスを計測し，比較を行った．その結果，提案手法を適用することによって多くのモジュールでメトリクス値が変化し，理解性の低いモジュールをより効率的に特定できることを確認できた．

続いて，2. について説明する．Buse と Weimer の調査結果では，ソースコード可読性に最も影響を与えている要因は識別子の数であった [5]．また，変数が定義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もある [18]．このような状況を改善するためには，変数のスコープを狭める，変数の代入文を参照される直前まで移動するといった文の並び替え操作が有効である．ソースコードの可読性を向上させるためのリファクタリング手法として，空行の挿入やインデントの整形などが行われている [7, 19]．しかし，文の並び替えを行おうとする場合，振る舞いを保つためには文の間にある制御依存，データ依存，実行順序といった様々な関係を考慮する必要がある．従って，文を自動で並び替えることは容易ではなく，このような研究はこれまでに行われていない．本研究では，変数の定義・参照間の距離に着目して，モジュール内の文を並び替える手法を提案する．提案手法をオープンソース・ソフトウェアに適用したところ，提案手法によって並び替えの行われたモジュールは可読性が向上したという結果が得られた．

2　準備

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いわれている [1]．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 [20]．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，動作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，ソースコードを読み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 [2, 3, 4]．

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 [21, 22]．

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ている．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．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 [23].

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 [24]．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 [6]．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いわれている．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 [25].

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 [26]．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 [9].

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 [9]．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 [9]．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いわれている．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 [8]．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 [10, 12, 27]．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（コードクローン）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 [28, 29]．

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いわれ，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，クラス内の変数及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いえる．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 [30].

メソッドの抽出

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 [30].

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 [31]．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 [30]．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 [32].

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 [10, 11, 33]．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 [34, 35]．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いえる．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 [13]．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

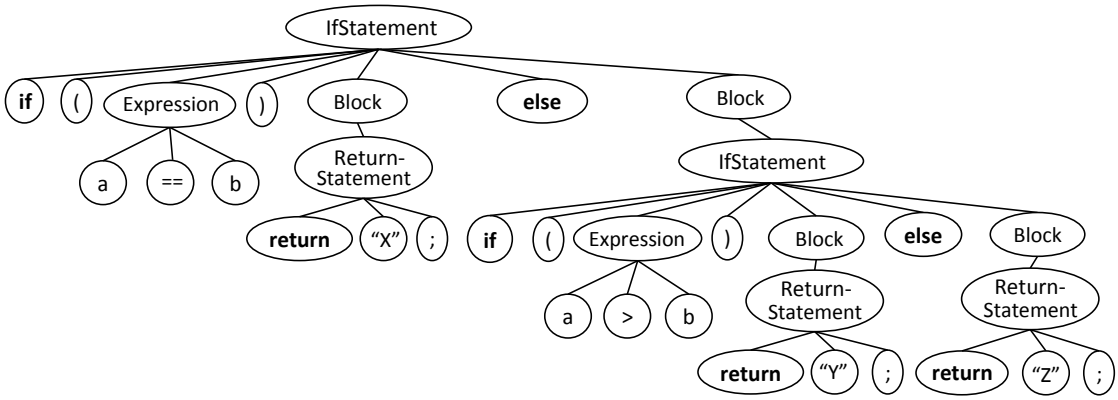
CK メトリクスは，オブジェクト指向デザインにおいてクラス構造に基づく複雑度を評価するためのもので，Chidamber と Kemerer によって提案された [14]．クラスの重み・結合度・凝集度・応答数・継承の深さ・子クラスの数の計 6 つのメトリクスが定義されている．CK メトリクスは，特にフォールトプローンモジュールの検出に有効であるといわれている [36]．また，クラスの凝集度を計測することでクラス抽出などのリファクタリング候補を特定することができるが，提案されている手法の多くは CK メトリクスに含まれる凝集度や，それを元に新たに提案されたメトリクスを用いている [37]．

2.4 抽象構文木

抽象構文木 (Abstract Syntax Tree, 以下 **AST**) とは，ソースコードの構文情報を木構造で表したものである．抽象構文木の例を図 1.1 に示す．

```
if (a == b) {
   return “X”;
} else if (a > b) {
   return “Y”;
} else {
   return “Z”;
```

(a) ソースコード



(b) AST

図 1.1: AST の例

AST 上のノードは構文上の 1 つの要素を表し，エッジで直接結ばれた子にあたるノードは，その詳細情報を表している．例えば，AST 上のルートにあたる $IfStatement$ の構文情報は，その子であるノードによって表されており，その内容は **if** ($Expression$) $Block$ **else** $Block$ であることが分かる．このように，共通の親を持つノードを**兄弟ノード**という．AST において，兄弟ノードはソースコード上の出現順に並んでいる．また，本論文で用いる AST では，ソースコード中の中括弧が表すような，複数の文を並べることができる箇所を $Block$ というノードで表し，ソースコード上でもこれをブロックと呼ぶこととする．ただし，図 1.1(a) のように，else 節の直後が if 文のみである場合も，if 文の構文としては本来中括弧で括ることによって複数の文を取ることができるため，$Block$ ノードが存在することとする．

AST などの木構造を解析する際，ノードへの訪れ方として以下のものが存在する．

前順走査 (preorder traversal)

まずルートノードを訪れる．訪れたノードが子ノードを持てば，左から順番に前順走査する．例えば図 1.1(b) に対して前順走査すると，図 1.1(a) の 2 つの if 文は，その出現順通り訪れられる．

後順走査 (postorder traversal)

ノードが子を持てば，左から順番に後順走査する．最後にルートノードに訪れる．例えば図 1.1(b) に対して後順走査すると，図 1.1(a) の 2 つの if 文は，その出現順とは逆順に訪れられる．

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，識別子の数は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 [5]．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 [22]．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 [18]．この結果，キューにない変数を参照するとき，すなわち定義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 [4]．Biegel らは，Java ソースコード中のフィールド及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 [38]．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 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ることが報告されている [36]．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 [5]．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 [16, 39, 40]．

3.3 サイクロマチック数の計測方法

Jbara らは Linux カーネルに含まれるサイクロマチック数の非常に大きな関数を調査し，連続して記述されてた単純な if 文や switch 文中の case 文は，人が見て複雑だと感じる要因ではないことを示している [17]．サイクロマチック数を提案した McCabe 自身も，case 文を分岐として考慮しない方法について言及している．サイクロマチック数の計測方法には様々な改善案があり，上記のように case 文を分岐として考慮しないものや，条件節中の関係演算子の数を考慮にいれるものがある [41, 42]．また，Vinju と Godfrey は，ソースコード中に繰り返し記述された制御構造に着目して，制

御フロー上でそれらを圧縮した上でサイクロマチック数を計測し，ソースコード理解性との相関を調査した　[43]．調査結果の中で，制御構造の圧縮によってサイクロマチック数の値が下がったメソッドは理解しやすいものが多く，制御構造の繰り返しによって，サイクロマチック数は理解性を表すには大きすぎる値を取る可能性があると報告されている．ただし，ここで行われた理解性の評価は，著者らの主観的なものである．

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 [19]．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 [44]．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 [7].

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 [11]．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 [45]．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 [46].

第 2 部

メトリクス計測の前処理となるソースコード簡略化手法の提案と評価

1　研究動機

図 2.1(a) はあるソフトウェアに含まれるメソッドである．このメソッドには多くの if 文が含まれ，ネストの深い構造になっている．サイクロマチック数は 33 と高い値を持ち，複雑なメソッドであることが分かる．一方，このソフトウェアにはサイクロマチック数 112 のメソッドが存在する．図 2.1(a) のメソッドと比べてサイクロマチック数が約 3 倍であることから，このメソッドは非常に複雑な構造を持つと予想される．しかし，このメソッドの構造は図 2.1(b) に示すように，単純な if-else 文が繰り返し記述されているのみであり，理解性の低い構造であるとは考えにくい．

本研究では，図 2.1(b) のようにソースコード中に繰り返し構造が含まれる場合，サイクロマチック数が従来の計測値よりも低く計測されるよう，繰り返し構造を折りたたんでソースコードを簡略化するという前処理を提案する．提案手法では，抽象構文木 (AST) を用いてソースコード中の繰り返し構造を発見し，折りたたんで単一の構造として表現する．ソースコード中の全ての繰り返し構造を折りたたんで簡略化し，そのソースコードに対してサイクロマチック数などのメトリクス値を計測すると，従来の値よりも低く計測できる．これによって，図 2.1(a) のような繰り返し構造を含まないメソッドのサイクロマチック数が相対的に高くなるため，複雑な構造を持つメソッドを見つけやすくなる．

```
  1: public Object getValoreIndirizzobenefattotrans(...) {
       ...
       if() { ... if() { ... if() { ... if() { ...
124:      if (soggetto != null) {
             ...
128:         else
129:        {
130:            ArrayIterator iter = ...;
131:            if (iter!=null && iter.size()>0) {
132:                iter.reset();
133:                IfIndirizzo recapito = ...;
134:                if(...)
135:                    return ...;
136:                else return null;
137:            }
138:            else
139:                return null;
140:        }
141:      }
142:      else return null;
        } } } }
189: }
```

(a) ネストの深い構造を含むメソッド

```
  1: public int getColumnIndex(final String sColumnName)
  2: {
  3:    if (sColumnName.compareToIgnoreCase(...) == 0)
  4:        return NDX_TI_ID_TITOLO;
  5:    else if (sColumnName.compareToIgnoreCase(...) == 0)
  6:        return NDX_TI_ID_COMPAGNIA;
        ...
204:    else if (sColumnName.compareToIgnoreCase(...) == 0)
205:        return NDX_FI_DESC_FILIALE;
206:    return -1;
207: }
```

(b) 繰り返し構造を含むメソッド

図 2.1: サイクロマチック数の高いメソッドの例

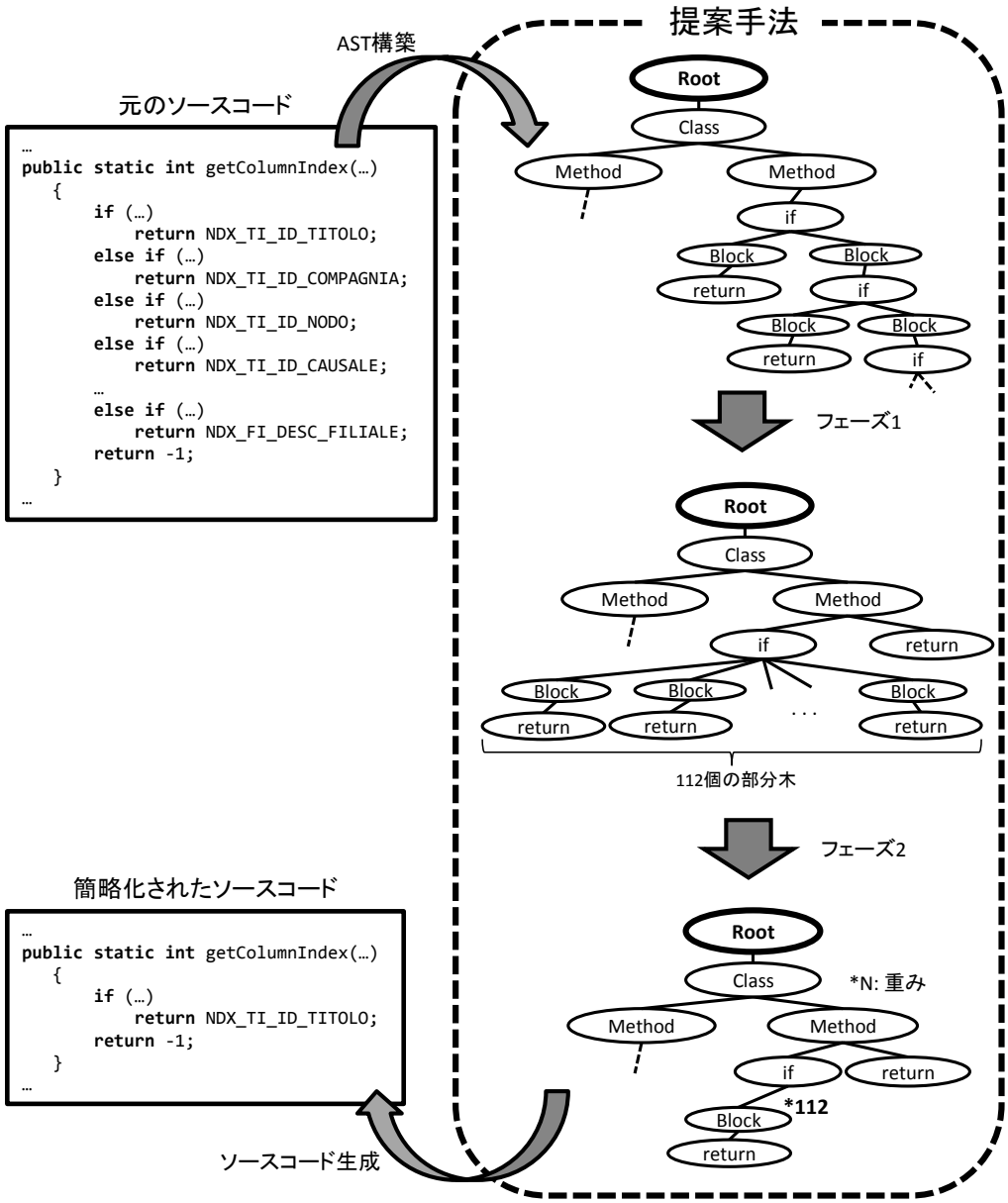


図 2.2: 提案手法の概要

2　提案手法

本手法は，ソースコードの AST を入力として，以下の手順で処理を行う．

フェーズ 1. AST の構造を変形する

フェーズ 2. 変形後の AST を後順走査し，繰り返し構造を特定と折りたたみを行う

フェーズ 2 が完了した時点で出力される AST からソースコードを生成すると，簡略化されたソースコードを生成することができる．図 2.1(b) のメソッドを含むソースコードを例とした手法の流れ

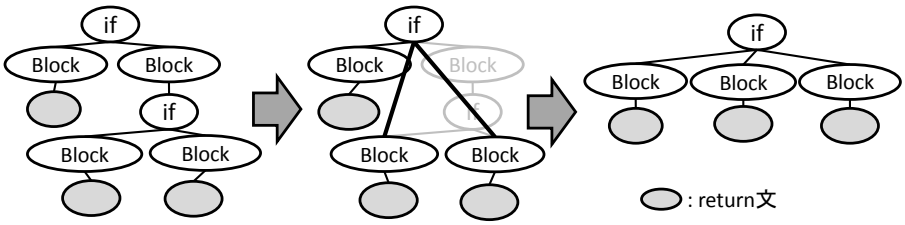


図 2.3: else-if 節の変形

を図 2.2 に示す．ただし，この図は AST 上の文またはブロックに関するノードのみを表記したものである．続いて，各フェーズについて説明する．

2.1 フェーズ 1. AST 構造の変形

フェーズ 1 では，AST 上で繰り返し構造を容易に特定できるようにするため，AST の変形処理を行う．例えば，図 2.2 では，メソッド先頭の if 文における else 節内に更に if 文が含まれる形が連なっている．しかし，このような if 文の階層構造は，多くのプログラミング言語においてソースコード上で並べて記述することが可能である．このように，ソースコードの見た目上連続している if 文の階層構造があれば，図 2.3 に示すような変形処理を行う．この変形処理とは，AST 上で else 節に続く *Block* ノードの直下に if 文を表すノードのみが存在する場合，これらのノードを削除し，削除された部分木の親ノードと子ノードを接続することを指す．

なお，本手法ではメソッド中の文構造にのみ着目し，変数名や条件式といった文の詳細は考慮しない．従って，これ以降は AST にそれらの情報を表示しない．

2.2 フェーズ 2. 繰り返し構造の特定と折りたたみ

フェーズ 1 で変形された AST に対して後順走査でノードを訪れ，訪れたノードの子ノード群に対して繰り返し構造を特定し，折りたたむという処理を行う．例えば図 2.2 では，*if* ノードに訪れたとき，子ノード群である *Block* ノードに対して，繰り返し構造かどうかの判定を行う．

以降，繰り返し構造の特定と，折りたたみ処理についてそれぞれ説明する．

2.2.1 繰り返し構造の特定

本手法では，繰り返し構造を特定する基準の 1 つとして，ノードが類似しているかどうか考える．この基準を表 2.1 に示す．表 2.1 中の”重み”については後述する．現在着目中の子ノード群（兄弟ノード）に対し，隣り合うノードが類似していれば，それらのノードからなる部分木全体を繰り返し構造とみなす．この基準によって繰り返し構造とみなされる例を図 2.4(a), 2.5(a) に挙げる．

また，ソースコード中には複数種類の文による記述の繰り返しが存在する．例えば以下のソースコードは，代入文とメソッド呼び出し文という 2 種類の文による記述の繰り返しである．

```
comparator = new ObjectIdentifierComparator();
cb.schemaObjectProduced( this, "2.5.13.0", comparator );
comparator = new DnComparator();
cb.schemaObjectProduced( this, "2.5.13.1", comparator );
```

このような複数種類の文による繰り返しも AST 上で繰り返し構造と判断できるように，兄弟ノードに着目したときの繰り返し構造の特定方法は以下の流れで行う．

1. 比較するノードの単位を 1 として，隣り合うノードが類似しているかどうか調べる
2. 比較するノードの単位を 1 増やし（このときノードの単位を n とする），n 個のノード単位で隣り合うノード列が類似するかどうか調べる
3. これ以上比較できなくなるまで，2. を繰り返す

複数のノード単位で繰り返し構造とみなされる例を図 2.6(a) に示す．この例の場合，隣り合うノード同士のみを比較した場合，文の種類が異なるため繰り返し構造とみなされないが，2 つのノード単位で比較したとき，種類の異なるノード列が互いに隣り合っているため，繰り返し構造とみなされることになる．

2.2.2 繰り返し構造の折りたたみ

繰り返し構造の折りたたみとは，繰り返し単位となる部分木（列）を 1 つだけを残して削除し，繰り返し回数を表す重みをルートノードに対して持たせることである．図 2.4(a), 2.5(a), 2.6(a) の繰り返し構造に対して折りたたみ処理を行った例を図 2.4(b), 2.5(b), 2.6(b) に示す．ただし，2.6(b) では 2 つのノード単位で重みを持たせている．

繰り返し構造の中には入れ子になったものも存在する．例えば以下のソースコードは，メソッド呼び出し文の繰り返しと，それをネスト内に含む if 文の繰り返しが存在する．

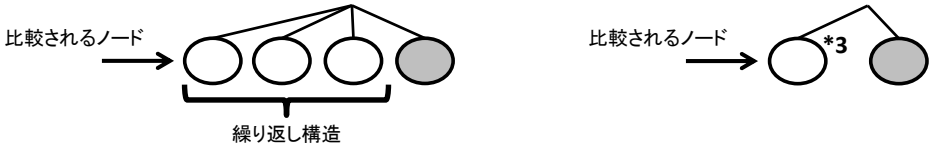
```
if (null != storepass) {
    cmd.createArg().setValue(“-storepass”);
    cmd.createArg().setValue(storepass);
}
if (null != storetype) {
    cmd.createArg().setValue(“-storetype”);
    cmd.createArg().setValue(storetype);
}
```

表 2.1: ノードの類似基準

比較されるノード対	類似とみなす判定基準
共に子を持たない	文の種類と重みが同じ
共に子を持つ	ノードをルートとする部分木が同形 かつ 対応する全てのノード対が類似

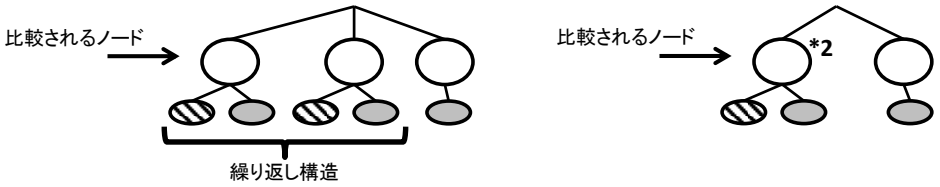
本手法では AST に対して後順走査を行っているため，2 つの if 文による兄弟ノードに訪れたとき，メソッド呼び出し文による繰り返し構造は既に折りたたまれている．従って，表 2.1 に示すよう，比較されるノード対が重みを持つ場合は，ノードの種類と重みが等しいときに類似と判断する．上記のソースコードに対する繰り返し構造の例を図 2.7 に示す.

ここに挙げた以外にも，ソースコード中には様々な繰り返し構造が存在することが述べられている [47]．本節で紹介したフェーズ 2 を用いると，[47] で紹介されたメソッド内部での繰り返し構造は全て折りたたまれることを確認した.



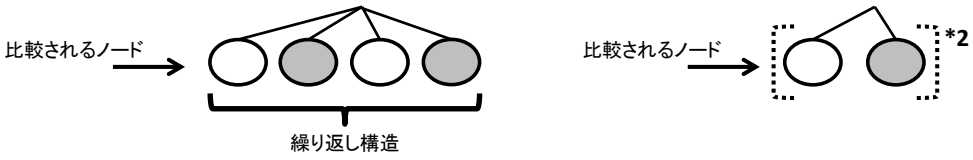
(a) 元の AST　　(b) 折りたたみ後の AST

図 2.4: 子を持たないノードによる繰り返し構造



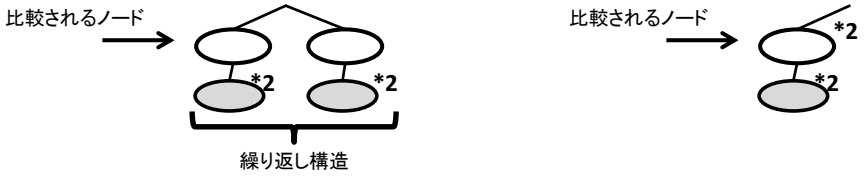
(a) 元の AST　　(b) 折りたたみ後の AST

図 2.5: 子を持つノードによる繰り返し構造



(a) 元の AST　　(b) 折りたたみ後の AST

図 2.6: 複数のノードによる繰り返し構造



(a) 元の AST　　(b) 折りたたみ後の AST

図 2.7: 繰り返し構造を要素に持つ繰り返し構造

3 実験

提案手法を実装し，評価実験を行った．この実験では Java で記述されたソースコードを対象とした．本実験の対象は，UCI source code data sets[48] に含まれる約 13,000 のソフトウェアである．このデータセットの規模を表 2.2 に表す．これらのソフトウェアに含まれる全てのメソッドに対して，提案する前処理を適用した場合としなかった場合のメトリクスを計測した．この実験では，次の 2 つの項目について調査し，提案手法の効果を確認する．

実験 1. 計測されるメトリクス値がどのように変化するか

実験 2. 計測されるメトリクスはソースコードの理解性と関係があるか

以降では，上記の項目に対する実験内容と結果について述べる．

3.1 準備

本実験で計測したメトリクスは，サイクロマチック数と行数である．サイクロマチック数は，ソースコード中の条件分岐を数え上げることで計測した．Java の場合，条件分岐とみなしたのは次の文である．

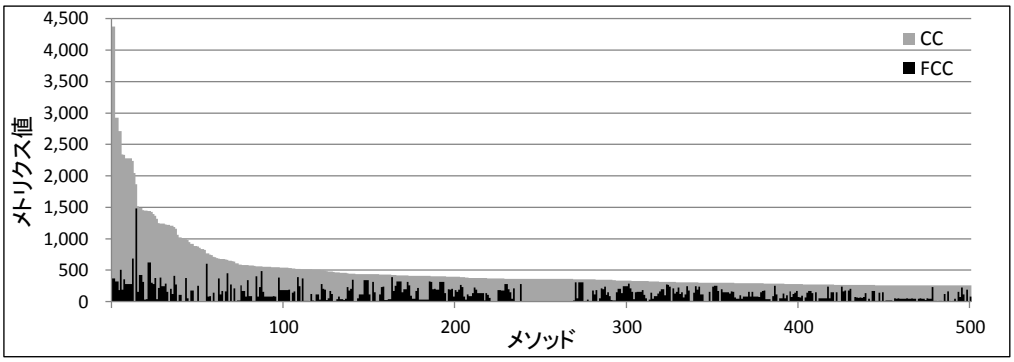
- if 文
- for 文
- 拡張 for 文
- while 文
- do-while 文
- switch 文中の case 文

表 2.2: UCI source code data sets の概要

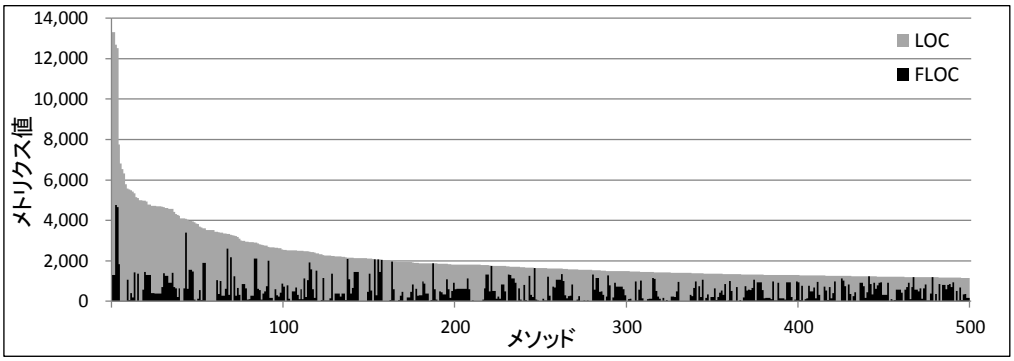
ソフトウェア数	13,193
メソッド数	18,366,094
総行数	361,663,992

表 2.3: 計測対象メトリクス

	サイクロマチック数	行数
提案手法を未適用	CC	LOC
提案手法を適用	FCC	FLOC



(a) サイクロマチック数



(b) 行数

図 2.8: メトリクス値の違い

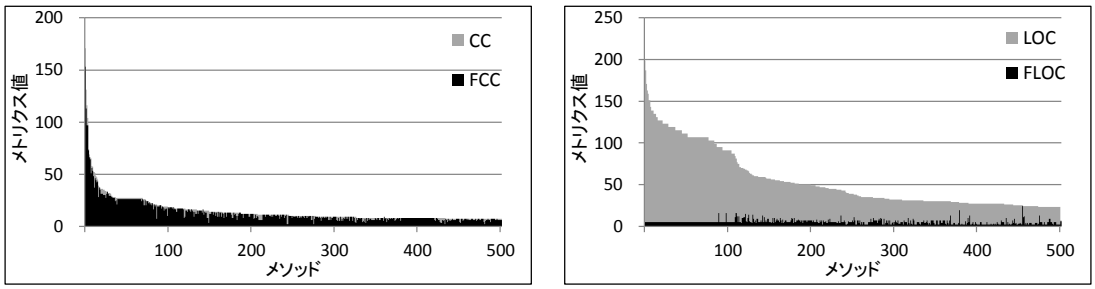
- try 文中の catch 節

また，行数は空行やコメント行を含まない値である．以降，これらのメトリクスを本論文では表 2.3 のように呼ぶ．

3.2 実験 1. 計測されるメトリクス値の比較

実験 1 では，提案手法適用の有無によるメトリクス値の違いを調査する．全メソッド中，CC, LOC それぞれの値が大きいメソッド上位 500 個について，メトリクス値の違いを表したグラフを図 2.8 に示す．2 つのグラフともに，CC, LOC の値に比べて FCC, FLOC の値が大きく減少しているメソッドが多く見られ，特に従来の計測値の大きかったメソッドはその変化が顕著である．従って，提案手法を適用することで，計測されるメトリクスの値は大きく影響を受けることが分かる．

また，このようなメトリクス値の違いをソフトウェアごとに調査したところ，図 2.9 に示すように，計測値にほとんど違いがないソフトウェアや，計測値が大きく異なるソフトウェアなど，ソフトウェアによって傾向が異なっていた．そこで，上記のような傾向をソフトウェアごとに得るため，各メトリクス値による上位 n 個のメソッド中に同じメソッドが含まれる割合 ”**一致率**” を定義し，ソフトウェアごとに計測した．サイクロマチック数の場合，一致率は以下のように定義される．



(a) 違いの少ない例　　(b) 違いの多い例

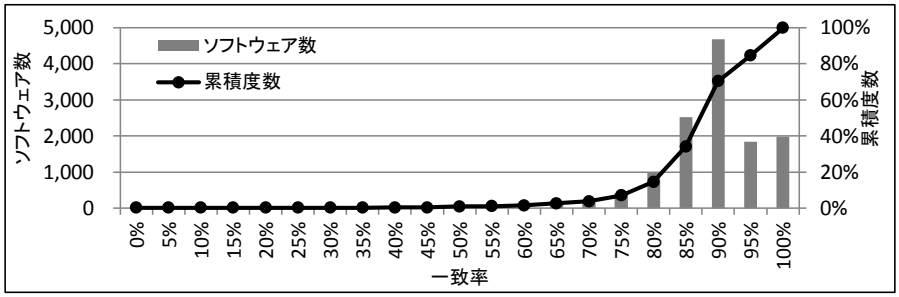
図 2.9: 各ソフトウェアにおけるサイクロマチック数の違い

$$\text{一致率}\ (n) = \frac{|CC(n) \cap FCC(n)|}{n}$$

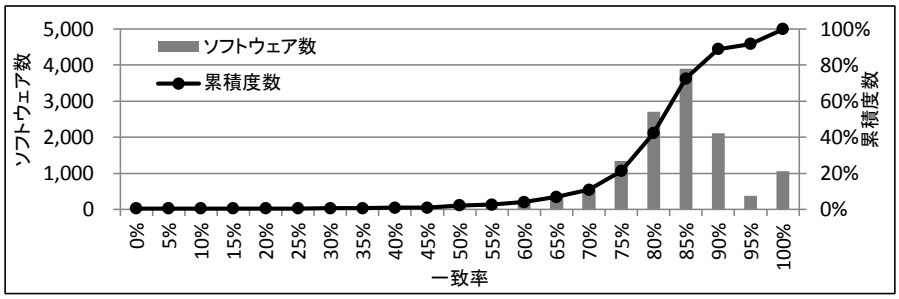
n は上位とみなすメソッドの数を表し，$CC(n)$ は CC の値による順位付け上位 n 個の集合を，$FCC(n)$ は FCC の値による順位付け上位 n 個の集合を表す.

全ソフトウェアに対する，サイクロマチック数，行数による一致率の分布を図 2.10 に示す．このグラフは一致率を 5%刻みで分布したヒストグラムである．例えば一致率 80%の項目は，一致率が 80%以上 85%未満であるソフトウェアの数を表している．また，対象ソフトウェアの規模が様々であるため，ここでは上位とみなすメソッド数 n をソフトウェアに含まれる全メソッド数の 20%にあたる数と定めている.

このグラフから，どちらのメトリクスについても一致率の高いソフトウェアが多いことが分かる．しかし一致率が 100%であるソフトウェアは，サイクロマチック数の場合はソフトウェア全体の約 15%，行数の場合は約 8%であり，提案手法によって多くのソフトウェアでメトリクス値上位のメソッドが変化するという結果を得られた.



(a) サイクロマチック数



(b) 行数

図 2.10: 一致率

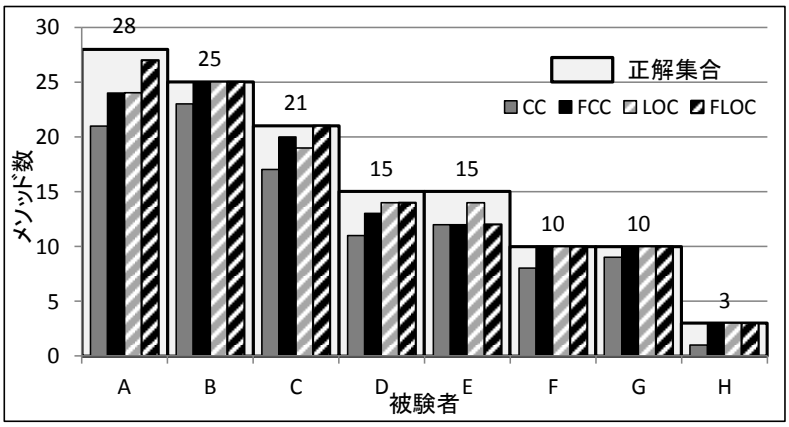


図 2.11: 上位 20 パーセント中の理解性の低いメソッド検出数

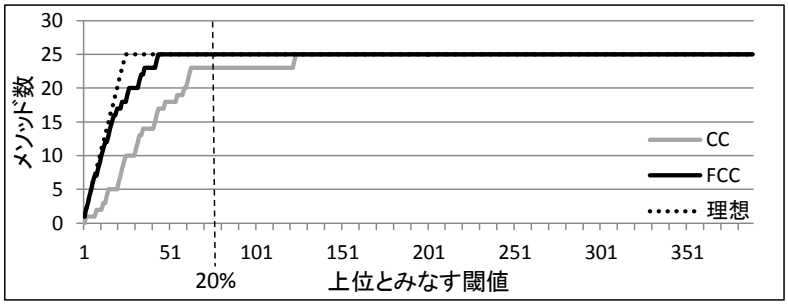
3.3 実験 2. ソースコードの理解性とメトリクス値の比較

実験 2 では，提案手法によって計測されたメトリクスが，ソースコードの理解性と関係があるかどうか調査を行った．本実験では，本学コンピュータサイエンス専攻の教員・大学院生 8 名を被験者とし，次の手順で実験を行った．

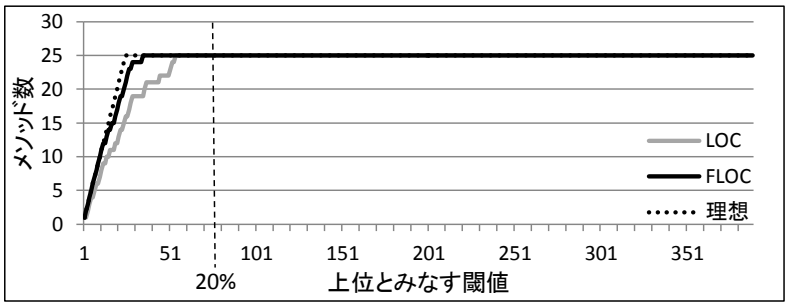
1. 対象ソフトウェア中のメソッド 1 つ 1 つについて，理解しづらいかどうかの評価を被験者に行ってもらう
2. 被験者によって理解しづらいと評価されたメソッドを**正解メソッド**とし，そのメソッドの集合を**正解集合**とする（正解集合は被験者の数だけ得られる）
3. 各メトリクス値によるメソッドの順位付け上位 20%を閾値とし，上位に含まれる正解メソッドの数を比較する

被験者に対してメトリクス値等の情報は提示されておらず，被験者は各メソッドについてソースコード閲覧によってのみ評価を行う．また，全メソッドを判断する被験者の負担が大きくならないよう考慮し，メソッド数 389 という規模がそれほど大きくないソフトウェア JCap を実験対象として選択した．このソフトウェアには，提案手法によって繰り返し構造とみなされ，計測されたメトリクス値が異なるメソッドが多く含まれている．各メトリクス値による順位付け上位 20%における一致率は，サイクロマチック数の場合は約 69%，行数の場合は約 67%である．

実験結果を図 2.11 に示す．例えば被験者 A の場合，正解メソッドの数は 28 個である．各メトリクス名が示す値は，そのメトリクス値上位に含まれる正解メソッドの数である．このグラフからは，サイクロマチック数，行数ともに，提案手法を適用した方が被験者 A に対する正解メソッドをより多く上位に含んでいることが分かる．すなわち，提案手法によって計測されたメトリクスは，被験者 A にとって理解性の低いメソッドを発見する際に有用であったといえる．全被験者の結果を見ると，



(a) サイクロマチック数



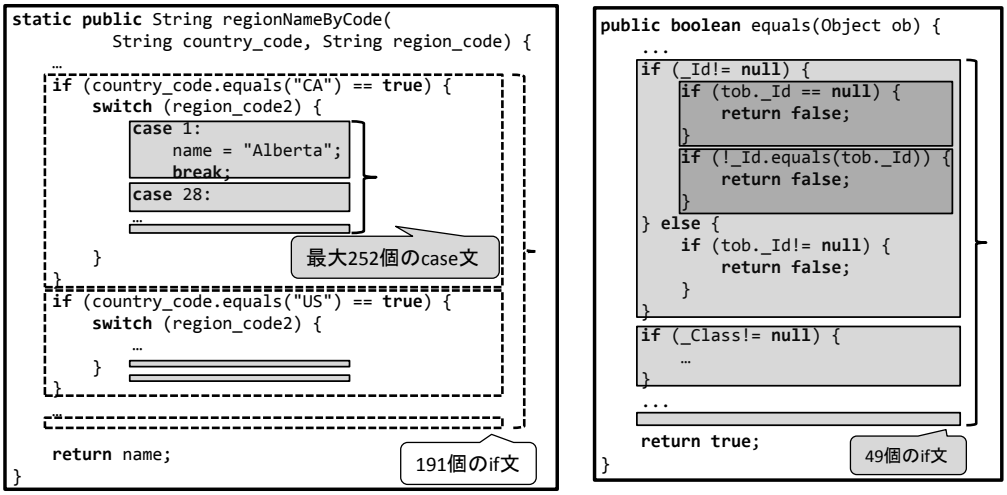
(b) 行数

図 2.12: 閾値の推移による理解性の低いメソッド検出数 (被験者 B の場合)

サイクロマチック数の場合は 8 人中 7 人の被験者について，提案手法を適用した方が各被験者に対する正解メソッドを多く上位に含んでいた．一方，行数の場合，このような傾向は 8 人中 2 人の被験者についてのみで，残りの 6 人中 4 人の被験者については，提案手法適用の有無に拘らず全ての正解メソッドを上位に含んでいたため評価が行えなかった．

続いて，上位とみなす閾値を変化させて上位に含まれる正解メソッド数を計測し，より詳細な分析を行った．図 2.12 は，各メトリクス値上位に含まれる被験者 B の正解メソッド数を，上位とみなす閾値ごとに表したグラフである．正解メソッドのみを上位に含む場合が理想であるため，各メトリクスによる正解メソッド数の推移がグラフ中の点線で示した形に近いほど，被験者によるソースコード理解性と関係があることを示している．グラフより，CC よりも FCC の方が，また LOC よりも FLOC の方が，被験者 B にとって理解しづらいメソッドを見つけた割合が高く，提案手法が有用であるといえる．特にサイクロマチック数で比較した場合，CC 値の上位 10 個中には 2 個の，FCC の上位 10 個中には 9 個の正解メソッドが含まれており，従来の計測手法よりも大きく改善できたといえる．

閾値の違いによる正解メソッド数の比較を他の被験者でも行ったところ，サイクロマチック数，行数ともに，8 人中 7 人の被験者に対して同様の結果が得られた．



(a) 図 2.8(a) に含まれるメソッド　　(b) 図 2.9(b) に含まれるメソッド

図 2.13: 発見された繰り返し構造

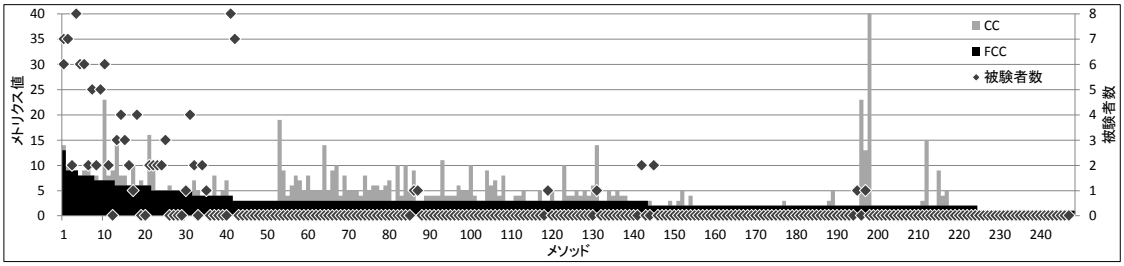
4 考察

本節では，提案手法によるメトリクス計測値の変化や計測値による順位の変化が妥当であったか考察を行う．

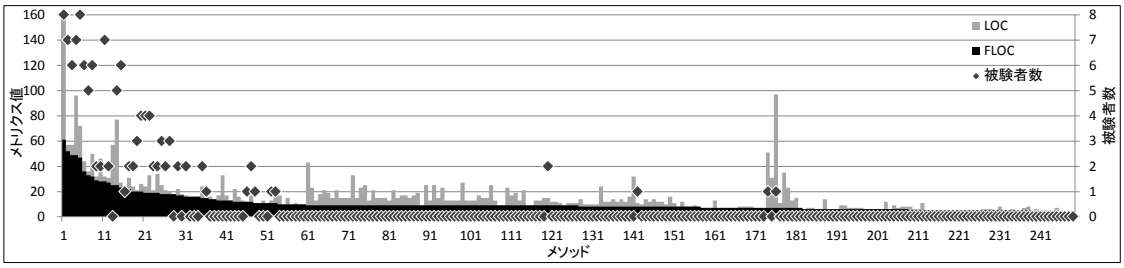
4.1 実験 1 について

まず，図 2.8(a) に含まれるメソッドの中から，最も CC の値が高いメソッドについて調査を行った．このメソッドは，CC 値が 4,377，FCC 値が 371 で，提案手法の適用によってサイクロマチック数が大きく減少している．このメソッドは図 2.13(a) に示すように，1 つの if 文の中に 1 つの switch 文を含むという構造が 191 個存在し，各 switch 文中に現れる case 文が最大で 252 回繰り返されているという巨大な構造であった．提案手法では，同じ構造の繰り返しでも繰り返し回数の異なるものは類似した構造とみなしていないため，case 文は折りたたむことができても if 文を折りたたむことはできなかった．各 if 文はネスト内に含まれる case 文の繰り返し回数以外は類似しているといえる．この例のように，case 文の繰り返し回数はあまり重要ではない可能性がある．

続いて，図 2.9(b) に含まれるメソッドについて調査を行った．このソフトウェア中で CC 値 100 以上のメソッドは全て，提案手法の適用によって FCC 値が 5 と計測されていた．最も高い CC 値は 199 で，図 2.13(b) に示すようにネスト内に if 文を 3 つ含む if 文が 49 回繰り返されたメソッドであった．また，このソフトウェアに含まれる FCC 値最大のメソッドは CC 値が 25，FCC 値が 24 で，CC 値による順位付けでは 455 番目に出現している．このメソッドは if 文や for 文による深いネスト構造を持っており，図 2.13(b) のメソッドと比べても非常に複雑な構造である．しかし，従来のメトリクス値による順位付けでは上位に提示することができない．従って，提案する前処理を適用することで



(a) サイクロマチック数



(b) 行数

図 2.14: メソッドのメトリクス値と正解とした被験者数の関係

理解しづらいメソッドの特定が容易になったといえる．

4.2 実験 2 について

実験 2 で対象としたソフトウェア中の全メソッドに対し，提案する前処理適用前後のメトリクス値と，そのメソッドを正解集合に含む被験者数を図 2.14 に示す．このグラフの x 軸では，提案手法適用後のメトリクス値が高い順にメソッドが並んでいる．このグラフからは，提案手法適用後のメトリクス値が高いほど，そのメソッドを正解集合に含む被験者数が多いという傾向が見られる．

全メソッド中，いずれかの被験者の正解集合に含まれるメソッドは 39 個であった．これらのメソッドを新たな正解メソッドとした場合，全メソッドは次の 4 種類に分類できる．

メトリクス値が高い正解メソッド

被験者 8 名全員から選ばれたメソッドは 2 つで，一方は行数，サイクロマチック数ともに高い値を示していた．もう一方は，GUI を構築するメソッドで、制御構造をあまり含まないためサイクロマチック数は低い値であったが，対象ソフトウェアの中で最も行数が長く，様々な機能を実装しているために単純な繰り返し構造ではなかった．

メトリクス値が高い非正解メソッド

提案手法適用後のメトリクス値が高いにも拘わらず，どの被験者からも選ばれなかったメソッドがいくつか存在した．例えば switch 文を 3 つ含み，各 switch 文中の case 文の数が異なるメソッドは，折りたたまれることなくメトリクス値が高いままであった．

5.1 節でも述べた通り，case 文の繰り返し回数による違いは，人が理解する上で重要ではない可能性がある．繰り返し回数による違いを吸収した折りたたみを行うと，このようなメソッドのメトリクス値を低く計測することができるため，今後はそのような改善が必要であると考えられる．なお，このような false-positive に相当するメソッド数が多い場合，被験者にとって理解しづらいメソッドを発見する際の妨げとなってしまうが，本実験対象ではこのようなメソッドは少なく，理解しづらいメソッドの特定に影響を与えるものではない．

メトリクス値が低い正解メソッド

単純な繰り返し構造を持つメソッドは，提案手法によって計測されるメトリクス値が特に低くなる．被験者の中には，このようなメソッドを理解しづらいとした例もあり，そのメソッドは従来のサイクロマチック数，行数ともに非常に高い値を持っていた．このことから，一定以上の繰り返し回数を持つ構造，あるいは従来の行数が長い構造は，ソースコードの理解性に影響を与える可能性があることが分かる．提案手法適用後のソースコードには，繰り返された回数や，提案手法適用前のソースコードから得られるメトリクス値といった情報が欠落している．そのため，提案手法適用後のソースコードから計測したメトリクスだけでなく，従来のメトリクスやソースコード可読性に関係する他のメトリクスと組み合わせて，理解性の低いソースコードを特定することが求められる．

メトリクス値が低い非正解メソッド

従来のメトリクス値が高く，折りたたみ後のメトリクス値が低いメソッドは，大部分が case 文による繰り返し構造であった．また，連続したメソッド呼び出しによる繰り返し構造も存在した．

5　結果の妥当性について留意すべきもの

実験 2 では提案手法による効果を測るために，繰り返し構造を含むメソッドがいくつか存在するソフトウェア 1 つを対象にした．しかし，対象ソフトウェアに含まれている繰り返し構造は，単純な case 文や if-else 節によるものが大部分であったため，複数の文からなる繰り返し構造や，再帰的に折りたたまれた構造に対する被験者の評価を得ることはできなかった．様々な繰り返し構造が含まれるソフトウェアを対象に実験を行えば，提案手法についてのより詳細な評価を得ることができると考えられる．

実験 2 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．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，多くの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．

6 まとめ

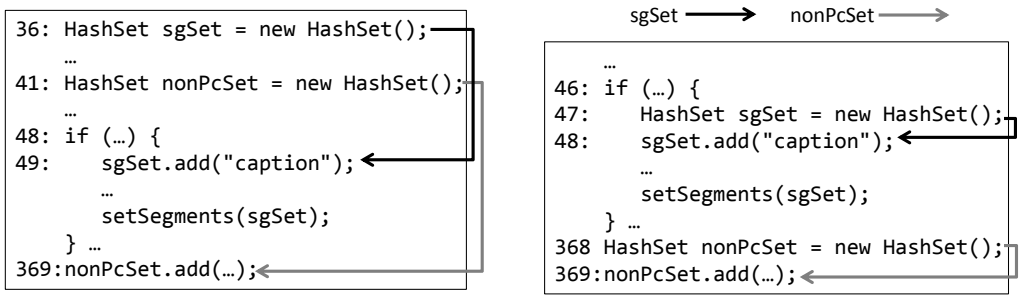
本研究では，従来のメトリクスを用いて理解性の低いソースコードを発見しようとする際，繰り返し構造によって値が増大してしまうという問題点に着目し，メトリクス計測の前処理となる繰り返し構造の折りたたみ手法を提案した．約 13,000 のオープンソースソフトウェアに対して，提案する前処理を適用した場合としなかった場合のメトリクス計測結果をメソッド単位で比較したところ，繰り返し構造が含まれるメソッドが多く存在し，多くのソフトウェアでその影響を受けるという結果を得た．更に，これらのメトリクスはソースコードの理解性と関係があることを確認した．

今後の課題は以下の 2 点である．

- 提案手法の改善を行う．例えば繰り返し回数による違いを吸収した上で折りたたみを行うことで，これまでと異なるメトリクスが計測される．
- 繰り返し構造の折りたたみ手法を応用したソフトウェア開発・保守支援を行う．例えば Eclipse などの統合開発環境上で，ソースコード中の繰り返し構造を可視化することで，ソースコードの可読性を向上させることができると考えられる．

第 3 部

可読性向上のためのプログラム文並び替え手法の提案と評価



(a) 移動前　　(b) 移動後

図 3.1: 文の移動例

1　研究動機

図 3.1(a) のソースコードは，変数の定義と参照が離れている例を示している．例えば，図 3.1(a) において，変数 $sgSet$ は 48 行目以降の if 文内でのみ参照されているにも拘わらず，外側のブロックで定義されている．一般的に，このように局所的に用いられる変数はスコープを狭めることが望ましい．変数 $nonPcSet$ については，定義・参照が同一スコープで行われているためスコープを狭めることはできないが，定義を行う文を参照される直前に移動することは可能である．これら 2 つの変数を定義する文について移動を行うと，図 3.1(b) のように変換することができる．

本研究では，ソースコードの理解性を向上させるために，文の並び替えを行う手法を提案する．図 3.1 の例に基づいて，具体的に次の 2 通りの移動戦略を用いる．

戦略 1 変数のスコープを狭めるため，内部ブロックへ文を移動する

戦略 2 変数の定義・参照間の距離を狭めるため，共通のブロック内で文を移動する

2 提案手法

2.1 変数の定義・参照関係の取得

本手法では，変数の定義・参照関係をデータフロー解析を用いて得られる Def-Use chain（以下，DUchain）から取得する [49]．DUchain とは，ある変数の定義・参照関係を表したものである．2 つの文 s_1, s_2 が次の条件を全て満たすとき，文 s_1 から文 s_2 へ DUchain が存在する．

- 文 s_1 は変数 v を定義する
- 文 s_2 は変数 v を参照する
- 文 s_1 から文 s_2 の間には，変数 v の再定義のない 1 つ以上の実行パスが存在する

ただし,「変数 v を定義する」とは，以下のいずれかの動作を表す

- 変数 v に対して代入を行う
- 変数 v が指すオブジェクトの状態を変更する

変数の定義・参照間の距離は，DUchain を元に計算を行う．ある DUchain c の距離 $distance(c)$ を，DUchain c によって結ばれた 2 つの文の間にある文の数とする．このとき，あるプログラムブロック B に含まれる全ての DUchain の総距離 $DataDistance(B)$ は，B に含まれる全ての DUchain の集合 $DUchain(B)$ を用いて以下の式で表す.

$$DataDistance(B) = \sum_{c \in DUchain(B)} distance(c) \tag{1}$$

2.2 手法の概要

本手法は，ソースコードを入力として以下の手順で文の並び替えを行う.

フェーズ 1. ソースコードから抽象構文木（AST）を構築する

フェーズ 2. AST を後順走査し，訪れたブロックに対して次の 2 つの移動戦略を適用する

1. 内部ブロックへ文を移動する
2. 共通のブロック内で文を移動する

フェーズ 3. 走査を終えた AST からソースコードを生成する

フェーズ 2 において，2 つ移動戦略が適用される様子を図 3.2 に示す．2 行目の if ブロックに訪れたとき，if ブロック内に移動することで変数のスコープを狭めることのできる文があれば，その文を if ブロック内へ移動する．続いて，if ブロックの $DataDistance$ が最小になるよう，ブロック内の文を並び替える．

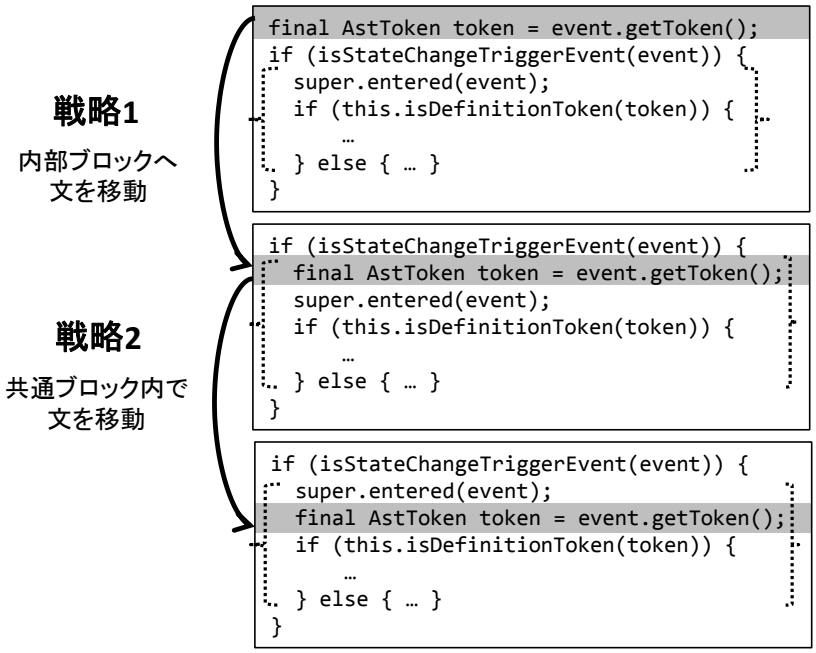


図 3.2: フェーズ 2 の適用例

2.3 並び替え戦略の実装方法

2.3.1 戦略 1: 内部ブロックへ文を移動する

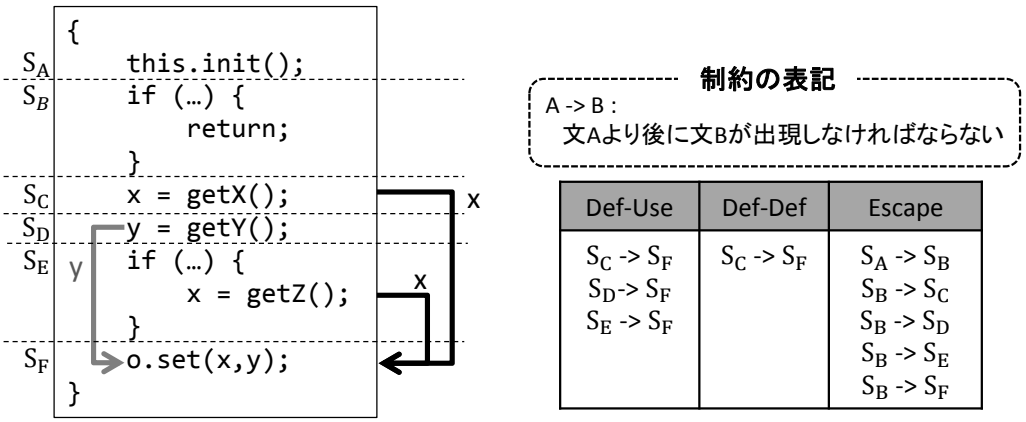
次の条件を満たすとき，文 s はブロック B 内へ移動可能であると定める．

- 文 s は変数宣言文である
- 文 s はブロック B の外側に存在し，ブロック B 内の文に対して DUchain が存在する
- 文 s はブロック B が実行される前に必ず実行される文である
- 文 s で定義された全ての変数は，ブロック B 内の文でのみ参照される
- 文 s で定義された全ての変数は，ブロック B が実行される前に再定義されることはない

上記の条件を全て満たす文が存在すれば，この文をブロック内の先頭要素として配置する．この戦略は文のスコープを狭めることのみを目的としているため，ここでは文をどの位置に配置すべきかということは考慮しない．

2.3.2 戦略 2: 共通のブロック内で文を移動する

着目中のブロック内の文を並び替える際，本手法では,「プログラムの振る舞いを変えない」という制約を満たす全ての文の並び（順列）を生成し，その中から $DataDistance$ が最小のものを選択



(a) 並び替え対象の文　　(b) 図 3.3(a) に対する順序制約

図 3.3: 文の順序制約の例

するという方法を用いる．

例として，図 3.3(a) のブロックに対して戦略 2 を適用する場合を考える．このとき，並び替え対象となる文は，文 S_A から S_F である．ただし，文 S_B や S_E のように，並び替え対象の文自身がブロックである場合も存在する．これらの文から生成される，プログラムの振る舞いを変えないための順序制約は図 3.3(b) の通りとなる．順序制約は下記の 3 種類である．

Def-Use 制約

ある変数の定義・参照関係にある 2 つの文は，現在の出現順序を保たなければならない．例えば図 3.3(a) の文 S_F は文 S_D から変数 y を参照しており，この 2 つの文を入れ替えると参照することができなくなってしまう．また，並び替え対象の文が，for 文のようなループ構造の中にあれば，変数を定義する文の方が参照する文より後に出現するということが考えられる．この場合も，現在の出現順序，すなわち参照する文から定義する文までという順序を保つ．

Def-Def 制約

同一名の変数を定義する文が複数存在するとき，その出現順を保たなければならない．例えば，図 3.3(a) の文 S_C, S_E ではともに変数 x が定義されており，文 S_F は，このどちらか一方から値を参照することになる．この 2 つの文を入れ替えると，文 S_C が文 S_E による変数 x の定義を完全に上書きしてしまうため，文 S_F は文 S_E で定義された値を参照することができなくなってしまう．

Escape 制約

ブロック外へのジャンプ命令を含む文は，その前後全ての文との出現順序を保たなければならない．例えば，図 3.3(a) の文 S_B は return 文を含んでいる．このとき，文 S_B より前の文は必ず実行

されるが，後の文は文 S_B の条件によっては実行されるとは限らない．なお，ブロック外へのジャンプ命令とは，return 文の他に continue, break 文や，Java 言語の場合 throw, assert 文が含まれる．

上記の制約を踏まえた上で，ブロック B 内の文に対する並べ替え方法を下記に示す．

1. 順序制約を満たす全ての順列を生成する

2. 各順列の中で，$DataDistance(B)$ が最小である順列を抽出する

しかし，$DataDistance(B)$ が最小である順列が複数存在する場合がある．その場合はさらに下記の処理を行う．

3. 抽出した各順列の中で，元の順列と最も近いものを 1 つだけ抽出する

本研究では，変数の定義・参照間の距離にのみ着目した文の並び替えを行うため，それ以外の要素で文の順序が入れ替わることは適切でないと考え，このような処理を行っている．この処理では，元の順列と最も近いものを選ぶために，Spearman の順位相関係数を利用する．Spearman の順位相関係数とは，2 つの順列がどのくらい似ているかを示すもので，全く同じであるとき 1.0 を，正反対であるとき -1.0 を示す．従って上記の処理では，2. で得られた全ての候補と元の順列との間で Spearman の順位相関係数を計測し，最も値が高いものを出力とする．

3 評価実験

提案手法を実装し，評価実験を行った．この実験では Java で記述されたソースコードを対象とした．

本実験の目的は，提案手法を用いた文の並び替えを行うことによって，ソースコードの理解性が向上するかどうか評価することである．実験対象として，オープンソースソフトウェアである TVBrowser を用いた．

3.1 準備

TVBrowser に含まれる全メソッドに対して，提案手法を適用したところ，215 のメソッドで文の並び替えが行われた．提案手法の評価を行うため，これらのメソッドからランダムに 20 個抽出し，オリジナルのソースコードと，提案手法を適用した後のソースコードとで，どちらが理解しやすいか被験者に答えてもらうアンケートを実施した．なお，これらのメソッドが提案手法適用後も振舞いを保っていることは手動で確認している．アンケートは以下の手順で行う．

1. 各メソッドからコメント・空行を取り除き，また，同じメソッドについてインデントや改行位置などのフォーマットを統一する
2. 各メソッドにつき，オリジナルと並び替え結果をランダムに A, B と区別し，読みやすさについて以下の項目から選んでもらう
   - A の方が読みやすい
   - B の方が読みやすい
   - 読みやすさに違いはない

上記のアンケートを web 上で公開し，被験者を募った．その結果，44 名の被験者が実験に参加した．アンケートでは被験者の Java 使用経験について質問を行っており，その結果は表 3.1, 3.2 の通りである．

表 3.1: Java を使ったプログラミング経験

内訳	人数
全く使ったことがない	1 名
1,000 行未満	7 名
1,000〜10,000 行程度	23 名
10,000 行以上	13 名

表 3.2: Java の使用機会（複数回答あり）

内訳	人数
授業（学生時代）	31 名
研究（学生時代）	28 名
趣味	18 名
仕事	12 名

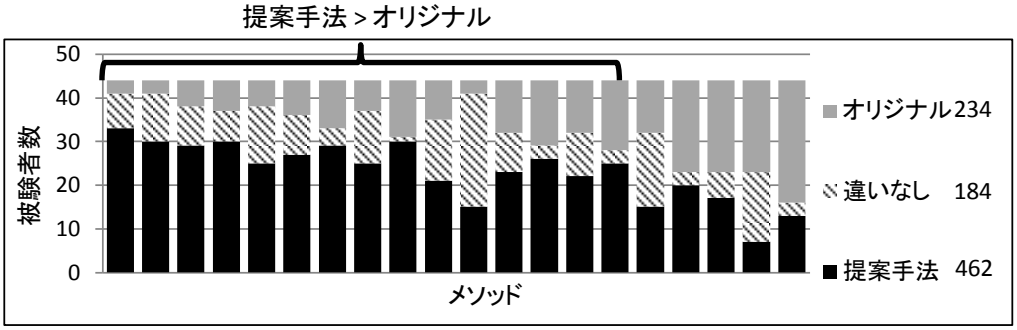


図 3.4: 各対象メソッドに対する回答の内訳

3.2 結果

実験結果を図 3.4 に示す．グラフ上の縦棒は各メソッドを表し，44 名の被験者が選んだ回答の内訳を表示している．また，各内訳の合計値をグラフの右側に記載している．グラフより,「違いがない」という回答を除けば，全 20 個中 16 個のメソッドで提案手法適用結果のメソッドを「読みやすい」と判断した被験者数が多いという結果が得られた．また，Wilcoxon の符号順位和検定より，提案手法を選んだ人数とオリジナルを選んだ人数には，p 値 0.002 で有意差があることを確認した．以上の結果から，提案手法によってメソッドの理解性を向上させる文の並び替えを行うことができたといえる．

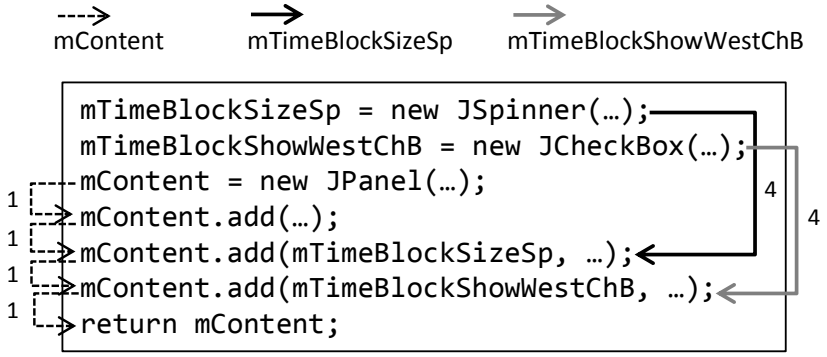
3.3 考察

図 3.4 より，4 つのメソッドについてはオリジナルのメソッドを「読みやすい」と判断した被験者が多いことが分かった．この 4 つのメソッドについて調査したところ，以下の特徴が見られた．

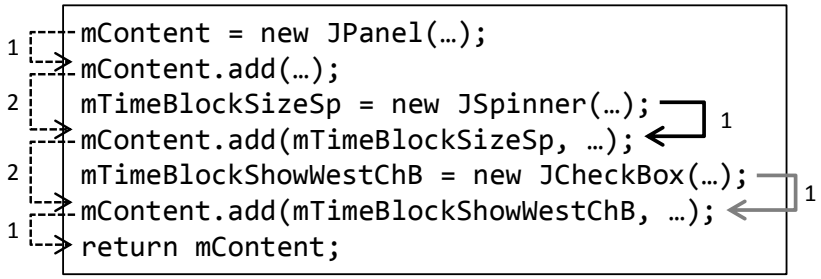
- 類似した変数名の宣言が連続して行われている
- 同名のオブジェクトによる，同名のメソッド呼び出しが連続して行われている

例えば，図 3.5 はオリジナルの方が読みやすいと判断した人数の最も多かったメソッドである．図中の矢印は DUChain を表し，そのラベルは DUchain の距離を表している．提案手法の適用によって，このメソッドの先頭で定義されていた 2 つの文が，参照される直前へ移動するという文の並び替えが行われた．しかし，被験者の多くはオリジナルの方が読みやすいと評価している．オリジナルのメソッドには，メソッドの先頭で定義されている 2 つの変数名が類似しているということ，変数 $mContent$ に対するメソッド呼び出し操作が連続して行われているという特徴がある．一方，提案手法適用後のメソッドでは，このような類似する文の並びが保たれていない．

提案手法による文の並び替えを行うことで，これらの類似した文の構造が保たれていない例がいくつか見られた．このことから，類似した文の並びはソースコードの理解性に貢献するものである可能性がある．



(a) オリジナル



(b) 提案手法適用後

図 3.5: オリジナルの方が読みやすいと判断されたメソッド

4　提案手法の拡張

実験の結果，ソースコードの理解性を向上させるために文の並び替えを行う場合，変数の定義・参照間の距離だけでなく，類似した文の構造を保持することを考慮に入れる必要性が見つかった．そこで，ソースコード中の類似した文の並びを識別し，その構造を保持したまま文の並び替えを行うという手法の拡張を行った．

4.1　類似した文の並びの識別

拡張手法では，第 2 部で紹介したソースコード簡略化手法のうち，繰り返し構造を識別する部分を用いて，類似した文の並びの識別を行う．繰り返し構造の定義は第 2 部 2 節で述べた通りである．上記の定義では，同じ種類の文であれば類似した文として判定しているが，実験の結果を踏まえて，次の 3 種類の文については類似と判定するために以下の条件も考慮する．なお，Java におけるこれらの文の記述形式を表 3.3 に示す．

メソッド呼び出し文

メソッド呼び出し文では，メソッド名と，メソッドが呼び出されるオブジェクトを指す変数名（以下，オブジェクト名）の両方がそれぞれ完全一致であるときに，類似する文とみなす．ただし，オブジェクト名が明記されていない場合は，メソッド名が完全一致であるときに類似する文とみなす．

変数宣言文

変数宣言文では，複数の変数が 1 つの文として宣言されることがある．このような変数宣言文は，他のどの変数宣言文に対しても類似しているとはみなさない．また，ただ 1 つの変数が宣言されている場合，型名が完全一致であり，かつ宣言された変数名が類似しているときに類似する文とみなす．変数名が類似しているかどうかの判定基準は後述する．

代入文

代入文では，左辺に現れる名前が類似していれば類似する文とみなす．

表 3.3: 文の種類と Java における記述形式

文の種類	メソッド呼び出し文	変数宣言文	代入文
記述形式	object.method(...); method(...);	type name; type name = ...; type name1, name2, ...; type name1, name2, ... = ...;	name = ...;

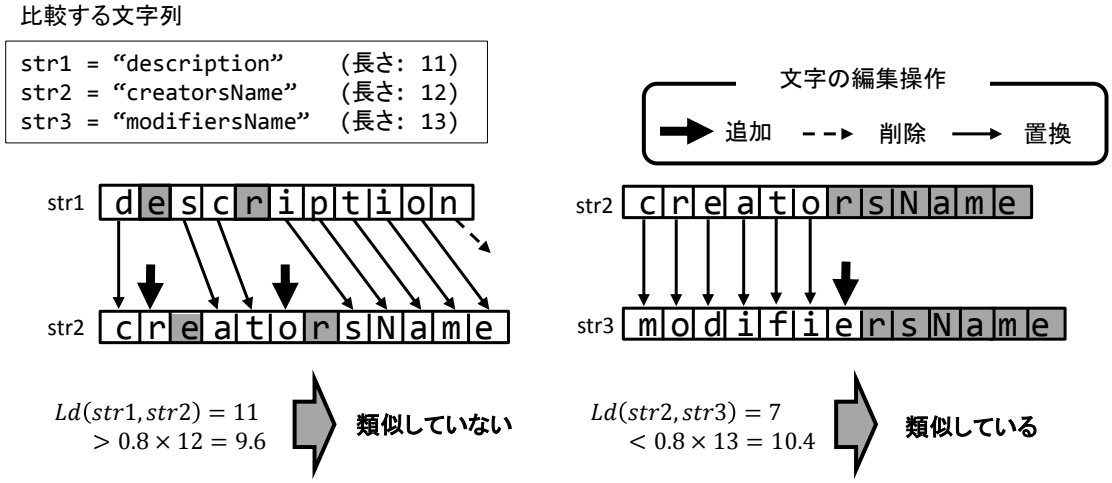


図 3.6: 文字列の類似判定の例

変数宣言文と代入文については，名前が類似するかどうかの判定を行う．この判定にはレーベンシュタイン距離を用いた．レーベンシュタイン距離とは，文字列のペアがどの程度異なっているかを表す指標である．一方の文字列からもう一方の文字列に変換する際に必要な文字の編集操作回数がレーベンシュタイン距離にあたる．文字の編集操作とは，文字の追加，削除，置換である．

拡張手法では，文字列のペア $str1$, $str2$ 間のレーベンシュタイン距離 $Ld(str1, str2)$ が次を満たすとき，文字列のペアは類似するとみなす．

$$Ld(str1, str2) \leq threshold \times max(len(str1), len(str2)) \tag{2}$$

ただし，$len(str)$ は文字列 str の長さを表し，$max(len(str1), len(str2))$ は，2 つの文字列のうち長い方の長さを表す．$threshold$ は 0 から 1 の値を取り，文字列を変換するために必要な文字の編集操作を許す割合の閾値である．今回は $threshold$ を 0.8 に設定する．

レーベンシュタイン距離を用いた文字列の類似判定の例を図 3.6 に示す．また，拡張手法によって識別される，ソースコード中の類似した文の並びを，以降は**類似構造**と呼ぶ．

4.2 拡張手法の確認

3 節でアンケート対象として用いた 20 のメソッドに対して拡張機能を加えた提案手法を適用した．その結果，20 のメソッドのうち 2 つのメソッドで，並び替えが行われなくなった．それらのメソッドは，3 節での実験において，オリジナルの方が読みやすいと評価された 4 つのメソッドに含まれるものであった．

4.3 追加実験

類似構造がソースコード理解に貢献するものかどうか調査するため，拡張手法を用いた追加実験を行った．実験対象として，オープンソースソフトウェアである Apache Triplesec 及び Apache Nutch を用いた．

4.3.1 準備

実験対象のソフトウェアに，類似構造を考慮した場合としなかった場合の提案手法を適用し，並び替え結果を得た．このうち，考慮した場合としなかった場合両方で文の並び替えが行われ，かつその結果が異なっていたメソッドの中から 5 つ抽出し，2 つの並び替え結果のどちらが理解しやすいか被験者に答えてもらうアンケートを実施した．なお，これらのメソッドが提案手法適用後も振舞いを保っていることは手動で確認している．アンケートは以下の手順で行う．

1. 各メソッドからコメント・空行を取り除き，また，同じメソッドについてインデントや改行位置などのフォーマットを統一する
2. 各メソッドにつき、類似構造を考慮した場合の結果を A，考慮しなかった場合の結果を B と区別し，読みやすさについて以下の項目から選んでもらう
   - A の方が理解しやすい
   - B の方が理解しやすい
3. 文の並びとソースコードの読みやすさについての考えについて，自由記述で答えてもらう

上記のアンケートを web 上で公開し，被験者を募った．その結果，31 名の被験者が実験に参加した．このうち，前回のアンケートにも参加した人数は 18 名である．被験者の Java 使用経験は表 3.4, 3.5 の通りである．また，上記の 3. については回答を任意としている．

4.3.2 結果

実験結果を図 3.7 に示す．グラフ上の縦棒は各メソッドを表し，31 名の被験者が選んだ回答の内訳を表示している．また，各内訳の合計値をグラフ右側に記載している．グラフより，メソッド m_B

表 3.4: Java を使ったプログラミング経験

内訳	人数
1,000 行未満	8 名
1,000〜10,000 行程度	20 名
10,000 行以上	3 名

表 3.5: Java の使用機会（複数回答あり）

内訳	人数
授業（学生時代）	24 名
研究（学生時代）	22 名
趣味	14 名
仕事	5 名

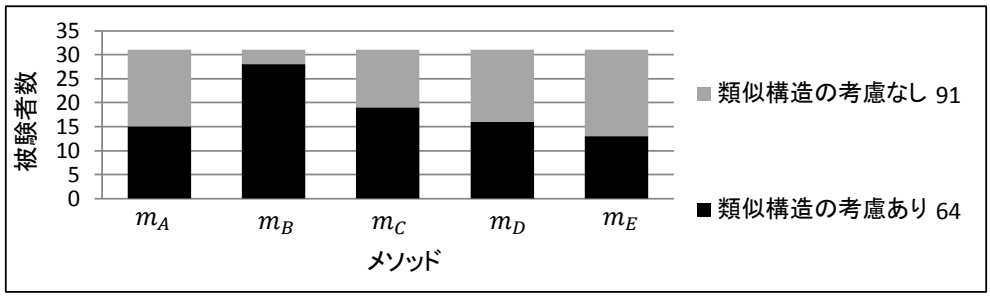


図 3.7: 各対象メソッドに対する回答の内訳

については，多くの被験者が類似構造を考慮した並び替え結果の方を理解しやすいと評価しているが，それ以外のメソッドについては意見が別れる結果となった．また，内訳の合計値を比べると，類似構造を考慮しない並び替え結果の方を理解しやすいと評価した被験者の数がやや多いという結果が得られた．

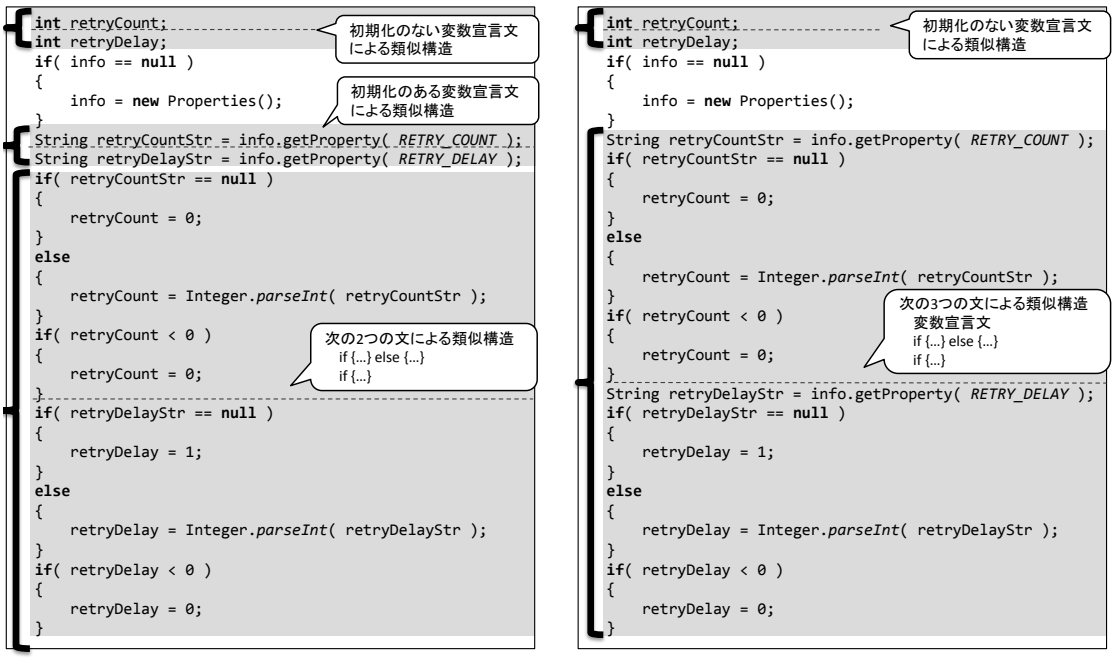
4.3.3 考察

本実験で対象とした 5 つのメソッドについて，それぞれ考察を行った．

メソッド m_A について

メソッド m_A に対する 2 通りの並び替え結果を図 3.8 に示す．メソッド m_A のオリジナルのソースコードからは，3 箇所の類似構造が特定された．この 3 箇所はいずれも 2 組の文（または文の並び）によって構成されており，1 つの類似構造の中に，他 2 つの類似構造で宣言された変数が出現している．類似構造を考慮した並び替えによって，図 3.8(a) に示すように，初期化のある変数宣言文による類似構造を，if 文による類似構造に近づけることができた．しかし，初期化のない変数宣言文による類似構造を近づけることはできなかった．この原因は，初期化されていない変数はどこからも参照することができないため DUchain が存在せず，提案手法によって並び替えを行う際に考慮されていないことであった．

また，類似構造を考慮しない並び替え結果では，図 3.8(b) に示すように，変数 $retryDelayStr$ の宣言文が，初めて参照される if 文の直前に移動していた．この移動によって，2 種類の類似構造であったものが，変数宣言文と 2 つの if 文からなる新たな類似構造となる結果となっている．被験者の何名かは，図 3.8(b) のように，変数の宣言と参照といったパターンが複数並んでいると理解がしやすいと答えている．先頭で宣言されている変数もこのパターンに組み込むことが可能だが，上記の通りこれらの変数宣言は移動が行えていない．このために図 3.8(a) の方が読みやすいと評価した被験者も多くなったのではないかと考えられる．



(a) 類似構造を考慮した場合

(b) 類似構造を考慮しない場合

図 3.8: メソッド m_A の並び替え結果

従って，類似構造を考慮する場合は，本手法のようにオリジナルのソースコード中に存在する類似構造のみを保持するのではなく，並び替えを行うことで新たに生成される類似構造も考慮に入れる必要がある．

なお，メソッド m_E についても，このような変数の宣言と参照というパターンが見られた．

メソッド m_B について

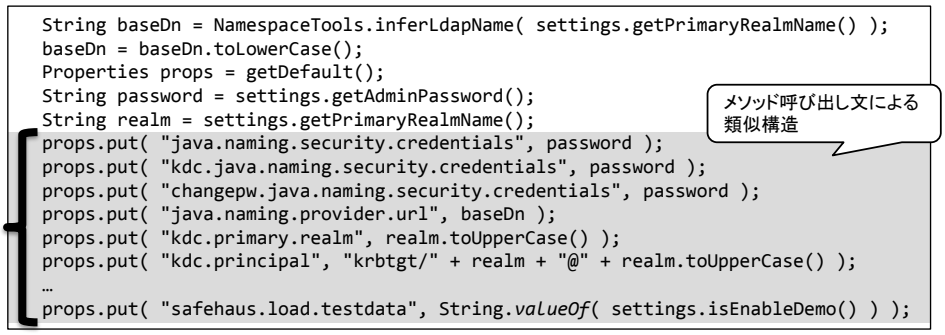
メソッド m_B のオリジナルソースコードからは，メソッド呼び出し文による類似構造が 2 箇所特定された．類似構造を考慮しない並び替え結果では，類似構造が 1 つも残らない結果となっている．このメソッドについては，大半の被験者が類似構造を考慮した並び替え結果の方が理解しやすいと評価している．この結果から，類似構造の存在はソースコードの理解性に貢献している可能性がある．

メソッド m_C について

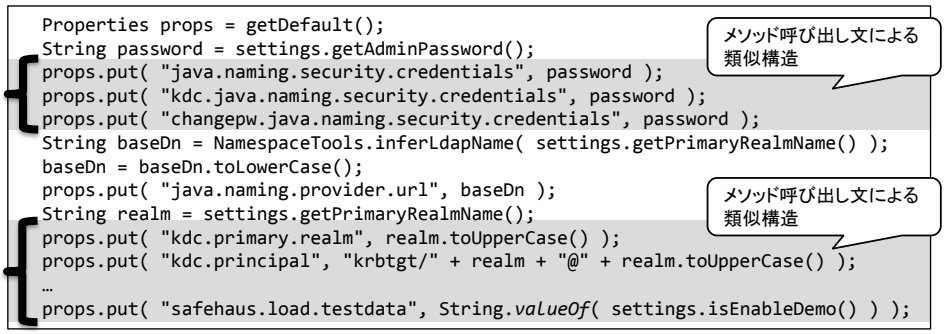
メソッド m_C に対する 2 通りの並び替え結果を図 3.9 に示す．メソッド m_C のオリジナルソースコードからは，12 個のメソッド呼び出し文による類似構造が 1 箇所特定された．このメソッドでは，先頭で 4 つの変数が宣言されており，そのうちの 1 つが指すオブジェクトに対してメソッド呼び出しが連続して行われている．類似構造を考慮した並び替え結果では，先頭の 4 つの変数を宣言する順序が入れ替わったのみであり，類似構造を考慮しない並び替え結果では，このうちのいくつかの変数が参照される直前まで移動されていた．

類似構造を考慮することによって，メソッド中に類似構造が存在しなくなったメソッド m_B とは異なり，メソッド m_C では類似構造をある程度保ちつつ変数の定義・参照間の距離を縮めることができている．自由記述に回答した被験者のうち数名から，類似構造の存在と変数の定義・参照間の距離が短いことはどちらも理解しやすいソースコードの特徴ではあるが，どちらを優先するかは場合によるといった回答が得られている．また，空行やコメントがあればより理解しやすくなると回答した被験者も何名かいた．実際に，このメソッドのオリジナルソースコードでは，連続したメソッド呼び出し中に 2 箇所空行が存在する．従って，類似構造をどの程度保持するべきかという基準は人によって様々な考えがあり，文の並び替えによって理解性向上の支援を行う際は，そのような意見を反映できるような環境が必要ではないかと考えられる．

なお，メソッド m_D についてもこのような並び替えが行われており，どちらも被験者による評価結果は分かれている．



(a) 類似構造を考慮した場合



(b) 類似構造を考慮しない場合

図 3.9: メソッド m_C の並び替え結果

文の並びと理解のしやすさについてのアンケート結果

自由記述による，文の並びと理解のしやすいについてアンケートには 15 名の被験者が回答した．回答結果については表 3.6 にまとめている．なお，表の各項目は全ての被験者の回答内容を漏れなく抽出したものであり，1 人の被験者からの回答が複数の項目に分かれていることがある．

自由記述の中で，8 名の被験者が類似した文の並びについて言及している．いずれも，類似構造が存在するとソースコードの理解がしやすくなるという内容であった．また，変数の定義・参照間の距離について言及した被験者は 8 名で，そのうち 7 名は，変数は参照される直前に定義されると良いと回答している．しかし，文の並び替えを行う際に，これらの要素を共に優先できるとは限らず，どちらを優先すると良いかは人によって異なったり，改行やコメントといった文の並び以外の要素に影響を受けたりということがある．

以上の結果から，文の並び替えは常に全自動で行うべきではないと考えられる．しかし，変数の定義・参照間の距離を小さくすることや類似構造を保つことで，ソースコードの理解性が向上することも確認できている．従って，文の並び替えを行う際は，本研究で提案する手法に加え，文の並びに関するユーザの意見を取り入れられるような仕組みが必要である．

表 3.6: 文の並びと理解のしやすさについてのアンケート結果

類似構造ついて

オブジェクトへの呼び出しをまとめると良い	4 名
変数の宣言と参照のパターンが連続していると良い	3 名
プロパティへの代入をまとめると良い	2 名
関連のある変数の定義をまとめると良い	1 名

変数の定義・参照間の距離について

変数の定義から参照までの距離を近づけると良い	7 名
変数の定義をまとめて行うと良い	1 名
定数は先頭で宣言すると良い	1 名

文の並びよりも重要な要素について

改行の位置が重要である	2 名
コメントが重要である	1 名
識別子の命名が重要である	1 名

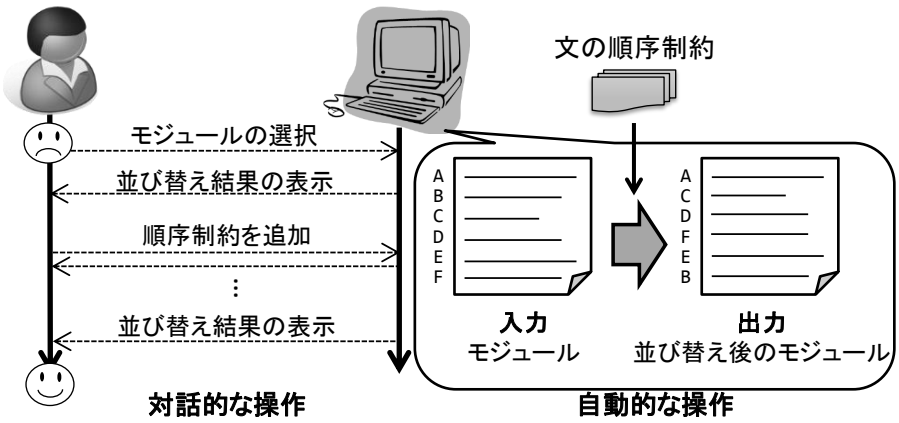


図 3.10: 対話的リファクタリング環境の概要

5 まとめ

本研究では，変数の定義・参照間の距離に着目して，ソースコード中の文を並び替える手法を提案した．1 つのオープンソースソフトウェアに対して手法を適用したところ，約 3,700 の並び替え可能なメソッドに対し，215 のメソッドを並び替えることができた．また，そのうちのいくつかのメソッドを対象に，提案手法を適用した場合としなかった場合のメソッドの理解性について 44 名の被験者による評価を得たところ，提案手法を適用することで多くのメソッドの理解性が向上したという結果が得られた．更に，理解性に影響を与える文の並び方には，変数の定義・参照間の距離だけではなく，類似した文の並びも影響を与える可能性があるという結果が得られた．

今後の課題としては，より多くのソフトウェアを対象に提案手法を適用し，文の並び替え手法を改善する必要がある．また，モジュールの理解性を向上させることを目的とし，文の並び替えを行う対話的なリファクタリングツールを構築する．この概要を図 3.10 に示す．ツール上でユーザが理解性を向上させたいモジュールを 1 つ選択すると，ツールは文の並びに関する初期の制約に従って文を並び替え，その結果を 1 つだけ提示する．初期の制約とは，プログラムの振る舞いを変えないために最低限守るべき制約であり，2 節で説明している．ユーザはその結果を確認し，ソースコードに反映させるか決定する．もし結果に満足できなければ，ユーザは文の並びに関する追加制約を指定し，もう一度実行する．追加制約には，例えば類似構造を保つことなどが含まれる．このような対話的な操作を繰り返すことで，最終的にユーザが望む並び替え結果を提示することができる．

謝辞

本研究を行うにあたり，理解あるご指導を賜り，常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して，有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究に関して，随時的確なご助言を頂きました 井垣 宏 特任准教授に深く感謝申し上げます.

本研究の全過程を通し，終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本研究に関して，多大なるご助言を頂きました，日本学術振興会特別研究員 畑 秀明 氏に深く感謝申し上げます.

本研究に関して，多大なるご助言，ご助力を頂きました，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博士後期課程 1 年の 堀田 圭佑 氏に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言，ご協力に心より感謝致します.

最後に，本研究に至るまでに，講義などでお世話になりましたコンピュータサイエンス専攻の諸先生方に，この場を借りて心から御礼申し上げます.

参考文献

[1] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, Vol. 34, No. 1, pp. 135–137, January 2001.

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

[3] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *Proceedings of the 19th European conference on Object-Oriented Programming*, pp. 33–48, July 2005.

[4] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. 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.

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

[6] Code conventions for the java programming language. `http://www.oracle.com/technetwork/java/codeconvtoc-136057.html`.

[7] X. Wang, L. 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.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[10] F. Simon, F. Steinbrúckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pp. 30–38, March 2001.

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

[12] Y. Higo, S. Kusumoto, and K. 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, November 2008.

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

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

[15] C. L. Goues and W. Weimer. Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 175–190, January 2012.

[16] Y. Shin, A. Meneely, L. Williams, and J. 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.

[17] A. Jbara, A. Matan, and D. G. Feitelson. High-mcc functions in the linux kernel. In *Proceedings of the 34th International Conference on Program Comprehension*, June 2012.

[18] M. Nakamura, A. Monden, T. Itoh, K. Matsumoto, Y. Kanzaki, and H. Satoh. Queue-based cost evaluation of mental simulation process in program comprehension. In *Proceedings of the 9th IEEE International Software Metrics Symposium*, pp. 351–360, September 2003.

[19] S. D. Jackson, P. T. Devanbu, and K. Ma. Stable, flexible, peephole pretty-printing. *Science of Computer Programming*, Vol. 72, No. 1-2, pp. 40–51, 2008.

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

[21] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp. 595–609, September 1984.

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

[23] S. Haiduc, J. Aponte, and A. Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 223–226, May 2010.

[24] Eclipse. `http://eclipse.org/`.

[25] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, Vol. 26, No. 11, pp. 861–867, November 1983.

[26] 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 Transaction of Software Engineering*, Vol. 27, No. 1, pp. 1–12, January 2001.

[27] E. V. Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*, pp. 97–106, October 2002.

[28] T. Kamiya, S. Kusumoto, and K. 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, July 2002.

[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pp. 96–105, May 2007.

[30] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, pp. 287–297, May 2009.

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

[32] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 233–243, June 2012.

[33] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 53–62, May 2011.

[34] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 452–461, May 2006.

[35] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pp. 200–210, June 2012.

[36] V. R. Basili, L. C. Briand, and W. 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.

[37] G. Bavota, A. D. Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, Vol. 84, No. 3, pp. 397–414, March 2011.

[38] B. Biegel, F. Beck, W. Hornig, and S. Diehl. The order of things: How developers sort fields and methods. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pp. 88–97, September 2012.

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

[40] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, and A. E. Hassan. Revisiting common bug prediction findings using effort-aware models. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pp. 1–10, September 2010.

[41] G. J. Myers. An extension to the cyclomatic measure of program complexity. *ACM SIGPLAN Notitces*, Vol. 12, No. 10, pp. 61–64, October 1977.

[42] pmccabe. `http://www.parisc-linux.org/~bame/pmccabe/overview.html`.

[43] J. J. Vinju and M. W. Godfrey. What does control flow really look like? eyeballing the cyclomatic complexity metric. In *Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 154–163, September 2012.

[44] D. C. Oppen. Prettyprinting. *CM Transactions on Programming Languages and Systems*, Vol. 2, No. 4, pp. 465–483, October 1980.

[45] P. A. Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *Proceedings of the International Symposium on Empirical Software Engineering*, pp. 53–62, November 2005.

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

[47] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Method and implementation for investigating code clones in a software system. *Information and Software Technology*, Vol. 49, No. 9–10, pp. 985–998, September 2007.

[48] C. Lopes, S. Bajrachaya, J. Ossher, and P. Baldi. Uci source code data sets. `http://www.ics.uci.edu/~lopes/datasets/`.

[49] U. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis: Theory and Practice.* CRC Press, 1st edition, 2009.