プログラム構造に着目した
ソースコード保守性の計測と改善に関する研究

提出先　　大阪大学大学院情報科学研究科

提出年月　2021 年 6 月

佐々木 唯

内容梗概

近年，情報通信技術の急速な発展により，ソフトウェアを中心とする情報サービス産業は拡大の一途を辿っている．それらのビジネスの中心的役割を担うソフトウェアは，社会の変化に合わせて迅速かつ確実に変更していく必要がある．ソフトウェアは最初に開発され納品された後，保守フェーズとして機能追加や不具合対応などが継続的に行われる．ソフトウェアのライフサイクル全体にかかるコストのうち保守コストの占める割合は高く，ソフトウェアの保守を効率化することが重要となっている．

ソフトウェアの保守フェーズでは，ソフトウェアの保守性を計測または改善する活動が度々行われる．保守性とはソフトウェアの品質特性の一つであり，たとえば再利用のしやすさ，修正すべき箇所の特定のしやすさなどの観点が含まれている．保守性はソフトウェアが持つ特性であるため，それを具体的に表す指標はない．そのため，保守性に関連する指標の計測方法や，保守性に影響を与える要因について，様々な観点から研究が行われている．たとえばソースコードの複雑さや可読性は，保守性に影響を与える要因であるといわれており，これらの指標を計測することで保守性の低い箇所を発見し，改善する活動が行われている．

本論文は，ソフトウェアの保守を効率化することを目指して遂行した，ソースコードの構造に着目して保守性を計測または改善する三つの研究について報告する．

一つ目は，ソフトウェアメトリクスを用いて複雑なモジュール，すなわち理解性の低いモジュールを検出する場面において，効率的に検出するためのソースコード簡略化手法の提案である．理解性の低いモジュールを検出する手段の一つに，サイクロマチック数や行数などのソースコードの複雑さを表すメトリクスを計測する方法が挙げられる．しかし，連続して記述された if 文のように，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複雑であるとは限らない．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ソースコード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．

二つ目は，ソースコードの可読性を向上させるための，プログラム文の並べ替え手法の提案である．ソースコードを読む作業は保守作業における開発者の行動の大半を占めるといわれている．また，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可読性に影響を及ぼすと考えられる．そこで，変数の定義と参照の間の距離に着目し

て，モジュール内の文を並べ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，並べ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，並べ替えの行われたモジュールは可読性が向上するという結果が得られた．

三つ目は，保守性の一つの観点としての，ソースコードに対する欠陥限局の適合性とその計測方法の提案である．欠陥限局とはソースコードに含まれる欠陥箇所を推測する技術であり，デバッグ支援技術の一つである．中でも近年，Spectrum-Based Fault Localization（SBFL）に関する研究が盛んに行われている．SBFL は，テストケースごとの成否と，どの文が実行されたかという情報を用いて，ソースコード中の欠陥箇所を推測する技術である．同一機能を実装したソースコードであっても，その記述や構造が異なれば，同じテストケースによって実行される文が変化するため，SBFL の精度に違いが生じる可能性がある．そこで，ソースコードが SBFL にどの程度適しているかを，そのソースコードに対する SBFL 適合性として提案し，更に SBFL 適合性の一つの評価指標として，SBFL スコアの計測手法を提案する．提案手法は，全てのテストケースを通過するソースコードに対し，ミューテーションテスト技術を活用して意図的に欠陥を発生させ，SBFL によってその欠陥箇所をどの程度正確に特定できたかを計測する．リファクタリングを題材に，ソースコード構造の違いによる SBFL スコアの違いを分析した結果，同一条件分岐先で実行される文の総数が，SBFL スコアに影響を与える要素であることを確認した．

業績リスト

関連発表論文

論文誌

[1-1] 佐々木唯, 石原知也, 堀田圭佑, 畑秀明, 肥後芳樹, 井垣宏, 楠本真二. 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. 電子情報通信学会論文誌 D, Vol. J96-D, No. 11, pp. 2634–2645, 2013 年 11 月.

[1-2] 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, 2014 年 2 月.

[1-3] 佐々木唯, 肥後芳樹, 柗本真佑, 楠本真二. プログラムに対する欠陥限局の適合性計測. 情報処理学会論文誌, Vol. 62, No. 4, pp. 1029–1038, 2021 年 4 月.

国際会議

[1-4] Yui Sasaki, Tomoya Ishihara, Keisuke Hotta, Hideaki Hata, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Preprocessing of Metrics Measurement Based on Simplifying Program Structures. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, Vol. 2, pp. 120–127, December 2012.

[1-5] Yui Sasaki, Yoshiki Higo, and Shinji Kusumoto. Reordering Program Statements for Improving Readability. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pp. 361–364, March 2013.

[1-6] Yui Sasaki, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. SBFL-Suitability: A Software Characteristic for Fault Localization. In *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*, pp. 702–706, September 2020.

その他論文

国際会議

[2-1] Yui Sasaki, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. Is Duplicate Code Good Or Bad? an Empirical Study with Multiple Investigation Methods and Multiple Detection Tools. In *Supplemental Proceedings of the 22nd annual International Symposium on Software Reliability Engineering (ISSRE 2011)*, November 2011.

謝辞

本研究活動全般において，常日頃より適切かつ丁寧なご指導を賜りました，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 楠本 真二 教授に心から感謝申し上げます．

本論文を執筆するにあたり，有益かつ的確なご助言を賜りました，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 井上 克郎 教授，増澤 利光 教授に心から感謝申し上げます．

本研究活動全般において，細部に渡り熱心かつ丁寧なご指導を頂き研究活動への活力を与えて下さいました，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 肥後 芳樹 准教授に心から感謝申し上げます．

本研究内容に関して，研究テーマの礎となる着眼点をはじめとした多大なる有益なご助言を頂きました，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 柗本 真佑 助教に心から感謝申し上げます．

本研究において，研究の主要な観点や発表資料に関して的確かつ有益なご助言を頂きました，信州大学工学部 岡野 浩三 教授に深く感謝申し上げます．

本研究において，随時議論の機会を設けてくださり研究内容の洗練にご助力を頂きました，大阪工業大学情報科学部 井垣 宏 准教授に深く感謝申し上げます．

本研究において，投稿論文の構成から研究結果の発展のさせ方まで多くの有益なご助言を頂きました，信州大学学術研究院工学系 畑 秀明 准教授に深く感謝申し上げます．

本研究活動において，具体的かつ有益なご助言と激励を下さいました，東京工業大学情報理工学院 林 晋平 准教授に深く感謝申し上げます．

本報告者の大阪大学大学院情報科学研究科博士前期過程在学中より，様々な場面で励まし支えて下さいました，当時事務補佐員の 神谷 智子 氏，藤野 香 氏，および現事務補佐員の 橋本 美砂子 氏に心より感謝申し上げます．また，在学中より研究内容の細部に渡って多大なるご助言とご助力を頂きました，堀田 圭佑 氏（現 富士通株式会社 富士通研究所）をはじめ，当時および現在の楠本研究室の皆さまに心より感謝申し上げます．

本研究の第 2 章における実験において，研究趣旨をご理解頂き被験者としてご協力頂きました，当時の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の教員および学生の皆さまに深く感謝申し上げます．また，本研究の第 3 章における実験において，研究趣旨をご理解頂き被験者としてご参加頂きました匿名の協力者の方々に深く感謝申し上げます．

本研究を大阪大学大学院情報科学研究科にて推進する機会を与えて下さいました，株式会社日本総合研究所 高野 宏治朗 取締役専務執行役員，宮奥 学 常務執行役員に心から感謝申し上げます．また，

本研究活動の意義を認めて下さり，本研究の遂行にあたり業務上の多大なるご配慮を頂きました，同社システム企画部 加藤 研也 部長，松田 卓久 次長，部員の皆さま，および，本研究の遂行にあたり多大なるご支援を頂きました，同社ＨＲマネジメント部 大島 貴明 部長，篠崎 宏州 次長，額宮 志織氏に心から感謝申し上げます．

最後に，本論文執筆に至るまでに，情報科学分野における勉学の場をご提供下さいました，大阪大学基礎工学部情報科学科，および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の諸先生方に，この場を借りて心から御礼申し上げます．

目次

第 1 章　はじめに　1

1.1　背景　1

1.1.1　ソフトウェアの保守性　2

1.1.2　ソフトウェア保守の効率化に向けて　3

1.2　研究の概要　5

1.2.1　ソースコード簡略化の前処理によるメトリクス計測方法の改善　6

1.2.2　プログラム文の並べ替えによるソースコードの可読性向上　7

1.2.3　ソースコードに対する欠陥限局の適合性計測　8

1.3　各章の構成　9

第 2 章　ソースコード簡略化の前処理によるメトリクス計測方法の改善　11

2.1　導入　11

2.2　研究動機　12

2.3　提案手法　14

2.3.1　準備　14

2.3.2　フェーズ 1：AST の構築　15

2.3.3　フェーズ 2：繰り返し構造の折りたたみ　16

2.3.4　フェーズ 3：ソースコード生成　20

2.4　実験　20

2.4.1　準備　20

2.4.2　実験 1：ソースコードの理解性とメトリクス値の比較　21

2.4.3　実験 2：計測されるメトリクス値の比較　24

2.5　考察　27

2.5.1　実験 1 について　27

2.5.2　実験 2 について　29

2.5.3　先行研究との比較　30

2.6　妥当性について留意すべき点　31

2.7　まとめと今後の課題　32

第 3 章　プログラム文の並べ替えによるソースコードの可読性向上　33
3.1　導入 . . . 33
3.2　研究動機 . . . 34
3.3　提案手法 . . . 35
3.3.1　変数の定義と参照間の距離の取得 . . . 35
3.3.2　手法の概要 . . . 36
3.3.3　戦略 A の実現方法 . . . 37
3.3.4　戦略 B の実現方法 . . . 37
3.4　評価実験 . . . 39
3.4.1　準備 . . . 39
3.4.2　結果 . . . 39
3.4.3　考察 . . . 40
3.5　提案手法の拡張 . . . 41
3.5.1　類似した文の並びの識別 . . . 41
3.5.2　拡張手法の確認 . . . 43
3.6　提案手法拡張後の評価実験 . . . 43
3.6.1　準備 . . . 43
3.6.2　二通りの並べ替え結果に対するアンケート結果 . . . 44
3.6.3　並べ替え結果に対する考察 . . . 44
3.6.4　文の並びと理解のしやすさについてのアンケート結果 . . . 47
3.7　妥当性について留意すべき点 . . . 49
3.7.1　被験者について . . . 49
3.7.2　実験で用いたプロトタイプについて . . . 49
3.7.3　メトリクス *totaldistance* について . . . 50
3.7.4　プロトタイプの実行時間について . . . 50
3.8　まとめと今後の課題 . . . 50

第 4 章　ソースコードに対する欠陥限局の適合性計測　51
4.1　導入 . . . 51
4.2　関連研究 . . . 53
4.3　SBFL 適合性 . . . 53
4.4　SBFL スコア . . . 55
4.4.1　SBFL スコアに影響する要素 . . . 55
4.4.2　SBFL スコアの算出方法 . . . 56
4.5　実験 . . . 58
4.5.1　準備 . . . 58
4.5.2　結果と考察 . . . 60

SBFL スコアが向上した例 61
SBFL スコアが低下した例 62
4.5.3 まとめ 65
4.6 妥当性について留意すべき点 66
4.7 まとめと今後の課題 66

第 5 章 むすび 69
5.1 まとめ 69
5.2 今後の研究方針 70

参考文献 71

図目次

1.1 ソフトウェア品質 3
1.2 新規開発と保守開発における作業コスト傾向（文献 [1] を元に作成） 4

2.1 サイクロマチック数の高いメソッドの例 13
2.2 提案手法の概要 14
2.3 AST の例 15
2.4 else-if 節の変形 16
2.5 子を持たないノードによる繰り返し構造 18
2.6 子を持つノードによる繰り返し構造 18
2.7 複数のノード（ノード列）による繰り返し構造 18
2.8 繰り返し構造を要素に持つ繰り返し構造 18
2.9 二種類の文による記述の繰り返し構造 19
2.10 入れ子になった繰り返し構造 19
2.11 AST を折りたたむ例 19
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) 22
2.13 メトリクス値上位に含まれる正解メソッドの数 23
2.14 メトリクス値の違い 25
2.15 各ソフトウェアにおけるサイクロマチック数の違い 25
2.16 メトリクス値上位 20% における一致率 26
2.17 メソッドのメトリクス値と正解とした被験者数の関係 27
2.18 メトリクス値が高い非正解メソッド 29
2.19 発見された繰り返し構造 30
2.20 差異を含む繰り返し構造 31

3.1 プログラム文の移動例 35
3.2 フェーズ 2 の適用例 36
3.3 文の順序制約の例 37
3.4 各対象メソッドに対する回答の内訳 40
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 41

3.6 文字列の類似判定の例 43
3.7 各対象メソッドに対する回答の内訳 45
3.8 メソッド m_A の並べ替え結果 45
3.9 メソッド m_C の並べ替え結果 47
3.10 文の並べ替えを利用した対話的なリファクタリング支援環境 48

4.1 構造の異なる二つのプログラムの SBFL 結果 54
4.2 SBFL スコア算出の流れ 56
4.3 ミュータントへの SBFL 実行結果 58
4.4 ケース 5：ガード節による入れ子条件記述の書き換え 61
4.5 ケース 1：変数の切り出し 63
4.6 ケース 3：重複した条件記述断片の統合 64
4.7 ケース 2：条件式の統合 65
4.8 ケース 4：制御フラグの削除 65

表目次

2.1 ノードの類似基準 16
2.2 文の種類（Java） 16
2.3 UCI source code data sets の概要 20
2.4 計測対象メトリクス 21
2.5 全ての正解メソッドの発見に必要なメトリクス値上位のメソッド数 23
2.6 適合率・再現率（被験者 A） 23

3.1 Java を用いたプログラミングの経験 40
3.2 Java の使用機会（複数回答あり） 40
3.3 文の種類と Java における記述形式 42
3.4 Java を使ったプログラミング経験 44
3.5 Java の使用機会（複数回答あり） 44
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 48

4.1 実験で用いるミューテーション演算子 59
4.2 対象リファクタリングと SBFL スコアの計測結果 60
4.3 条件分岐先ごとの文の分類（ケース 5） 62
4.4 条件分岐先ごとの文の分類（ケース 1） 63
4.5 条件分岐先ごとの文の分類（ケース 3） 64

第 1 章

はじめに

1.1 背景

ソフトウェアのライフサイクルにおいて，保守コストの占める割合は非常に高い．これまでの研究では，ライフサイクル全体の約 67% のコストが運用・保守段階で費やされているという報告があり [2]，典型的なコスト分布は，開発が 3 割，保守が 7 割ともいわれている [3]．ISO/IEC/IEEE 24765:2017 [4] において，ソフトウェアの保守は，ソフトウェアが納入された後，欠陥の訂正，性能やその他属性の向上，または変化した環境への適応のために修正するプロセスと定義されている[*1]．ソフトウェアは初めて納入されてからその役目を終え廃棄されるまで長い期間稼働し続ける．その間，ソフトウェアの実行環境であるオペレーティングシステムが古くなれば，バージョンアップに合わせてソフトウェアも修正し，組織や法令などに変更があればソフトウェアも対応しなければならない．設計時点で今後の変更が予測可能な場合は，何らかの形でパラメータ化して組み込むことができるが，実際に必要とされる多くの変更は，設計時点では思いもつかないものも多い [5]．特に，近年では情報通信技術が急速に発展し，従来の業種の壁を超えた新しい商品やサービスが次々と登場している．消費者の価値観も多様化しているため，このような変化をいち早く汲み取りソフトウェアを迅速かつ確実に変更できなければビジネスチャンスを失うことにもなりかねない．そのため，ソフトウェアの保守が効率的に行えるかどうかは，ソフトウェアを有する企業にとって死活問題であるともいえる．

ISO/IEC 14764:2006 [6]，及びそれに準ずる JIS X 0161:2008 [7] において，ソフトウェアの保守活動は次の四種類に分類される．

是正保守（corrective maintenance）
: ソフトウェア製品の引渡し後に発見された問題を訂正するために行う受身の修正．

予防保守（preventive maintenance）
: 引渡し後のソフトウェア製品の潜在的な障害が運用障害になる前に発見し，是正を行うための修正．

[*1] 原文は次のとおり．process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment

適応保守（adaptive maintenance）

引渡し後，変化した又は変化している環境において，ソフトウェア製品を使用できるように保ち続けるために実施するソフトウェア製品の修正.

完全化保守（perfective maintenance）

引渡し後のソフトウェア製品の性能又は保守性を改善するための修正[*2].

上記のうち，是正保守と予防保守はソフトウェアの問題に対する対応であり，「訂正」活動に分類される．一方，適応保守と完全化保守はソフトウェアの「改良」活動に分類される．ソフトウェア保守作業の多くは改良に関する活動であり，保守コストの 8 割以上を占めている [9].

保守作業においてソフトウェアのプログラムを修正する場合，修正内容による影響ができるだけ小さくなるよう，必要最小限の修正が検討されることがある．修正箇所が増えると，思わぬ箇所に欠陥を混入させてしまう可能性が高くなり，テストに必要な時間が増大するためである．しかし，修正を必要最小限とすることによって，時には不自然な実装や非効率的な実装が行われることがある．このような修正が，長い期間に渡って繰り返し行われることで，ソフトウェアは理解しづらく修正が困難な状態となることがある [10]．これは，ソフトウェアの保守性が低下していることを意味する．ソフトウェアの保守性とは，保守作業のためにソフトウェアを変更することの容易さを表しており，ISO/IEC 25010:2011 [11] ではソフトウェアの品質特性にも含まれている．保守フェーズでは，ソフトウェアの保守性が損なわれないよう管理することが大切である.

1.1.1 ソフトウェアの保守性

ISO/IEC 25010:2011 において，ソフトウェアの品質特性は図 1.1 の通り八つの特性で構成される．それぞれの特性は複数の副特性を持つ階層構造として整理されており，保守性は次の五つの副特性から構成される[*3].

モジュール性（Modularity）

一つの構成要素に対する変更が他の構成要素に与える影響が最小になるように，システム又はコンピュータプログラムが別々の構成要素から構成されている度合い.

再利用性（Reusability）

一つ以上のシステムに，又は他の資産作りに，資産を使用することができる度合い.

解析性（Analysability）

製品若しくはシステムの一つ以上の部分への意図した変更が製品若しくはシステムに与える影響を総合評価すること，欠陥若しくは故障の原因を診断すること，又は修正しなければならない部分を識別することが可能であることについての有効性及び効率性の度合い.

[*2] 旧規格である ISO/IEC 14764:1999 に準拠する JIS X 0161:2002 から引用した．JIS X 0161:2008 において，完全化保守は「引渡し後のソフトウェア製品の潜在的な障害が，故障として現れる前に，検出し訂正するための修正.」と定義されており，予防保守の定義と類似した表現となっている．情報処理推進機構（IPA）が発行する共通フレーム 2013 においても，完全化保守については旧規格の定義を掲載している [8].

[*3] ISO/IEC 25010:2011 に準ずる JIS X 25010:2013 [12] から引用.

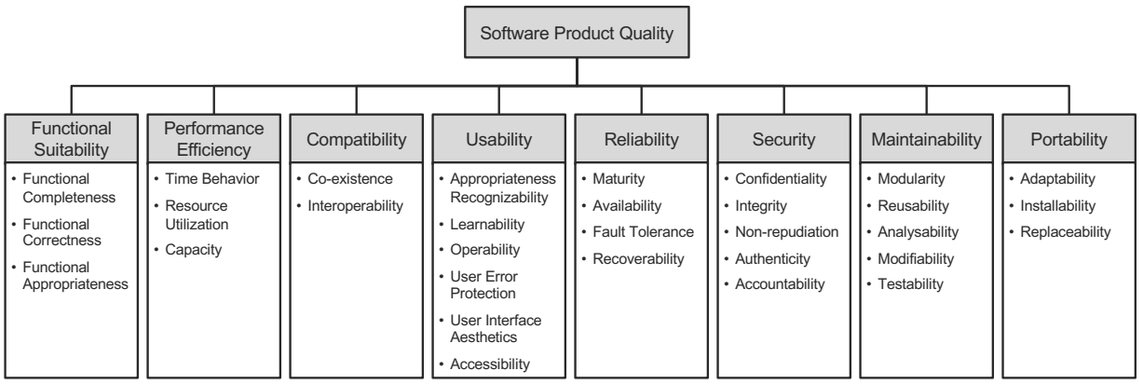


図 1.1　ソフトウェア品質

修正性（Modifiability）
: 欠陥の取込みも既存の製品品質の低下もなく，有効的に，かつ，効率的に製品又はシステムを修正することができる度合い．

試験性（Testability）
: システム，製品又は構成要素について試験基準を確立することができ，その基準が満たされているかどうかを決定するために試験を実行することができる有効性及び効率性の度合い．

保守性とはソフトウェアが持つ特性であるため，それを具体的に表す指標はない．そのため，保守性に関連する指標の計測方法や，保守性に影響を与える要因について，様々な観点から研究が行われている．Chen らは，ソフトウェアの開発プロセスに着目し，開発プロセス上の問題がソフトウェアの保守性に与える影響について調査した [13]．Matinlassi と Ovaska は，システム，アーキテクチャ，コンポーネントという三つの抽象化レベルから保守性を向上するための手段を述べている [14]．ソースコードに着目して，その特徴と保守性の関係を調査した研究も多く行われている．たとえば，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 [15]．また，ソースコードの複雑さや可読性も保守性へ影響を与える要因であるといわれている [16,17]．ソースコードはソフトウェアを構成する要素の中でも，目に見え，形に残る重要な成果物であるため，ソースコードの解析によってソフトウェアの特徴を把握し可視化する手法やツールは数多く存在する．保守フェーズにおいては，これらの技術を用いて継続的に保守性を計測および改善することが重要である．

1.1.2　ソフトウェア保守の効率化に向けて

ソフトウェア保守フェーズにおける開発作業（以降，保守開発）と，新規開発作業との違いの一つに，既に稼働しているソフトウェアをベースとした開発作業という点がある．保守開発では，現在のソフトウェアのアーキテクチャ，ソースコード，および運用環境などの制約を考慮する必要があるため，新規開発とは各工程の作業コストの傾向が異なる．増井らは，ソフトウェアの新規開発と保守開

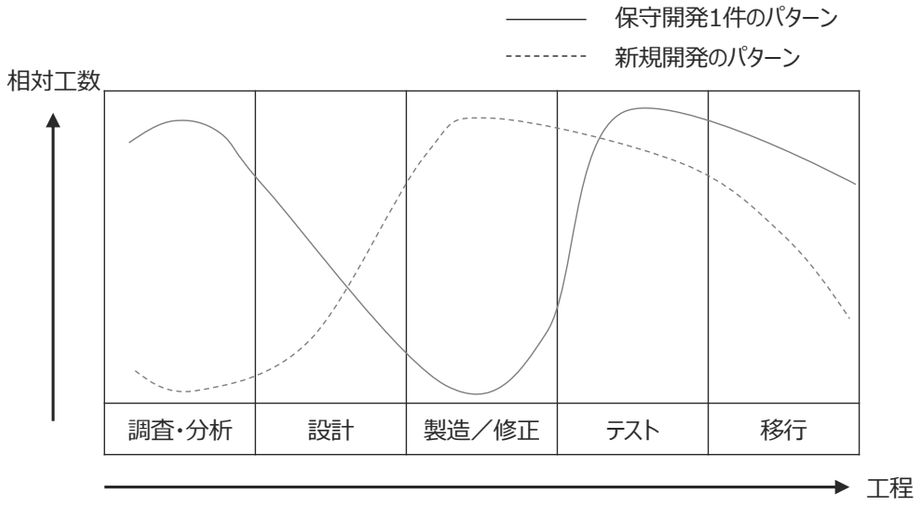


図 1.2　新規開発と保守開発における作業コスト傾向（文献 [1] を元に作成）

発における各工程の作業コストの割合を，図 1.2 の通り表現した [1]．図 1.2 によると，新規開発における工程ごとの作業コストは，調査・分析，設計と工程が進むにつれて増大し，製造工程でピークとなる．それ以降は減少する．一方，保守開発の場合，調査・分析工程とテスト工程では相対的に作業コストが高く，製造工程では相対的に低い傾向にある．これは，製造工程で実際に手を入れる部分がソフトウェアの一部であるのに対し，調査・分析工程ではソフトウェア全体の中から修正すべき箇所を検討する必要があること，テスト工程では，既存の機能が問題なく稼働することをソフトウェア全体に対して確認しなければならないことなどが影響している．

ソフトウェアライフサイクルの大半を占める保守活動においては，どのような作業にどの程度のコストが必要かを正しく理解した上で，効率的に保守をしていくことが求められる．ソースコードの計測および改善に着目した場合，調査・分析工程，およびテスト工程において保守を効率化するためには次の活動が効果的と考えられる．

メトリクスの計測の改善

ソフトウェアの保守作業を始める際，始めにソフトウェアのどの箇所をどのように修正するかを確認する必要がある．たとえば，保守性改善のためにリファクタリングを実施する場合，まずリファクタリングすべき箇所の特定が行われる．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り，不吉なにおいに対するリファクタリング方法が Fowler によってまとめられている [18]．また，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ため，メトリクスを用いてソフトウェアを計測し，品質の低い箇所を自動的に特定する活動も行われている [19–21]．ソフトウェアメトリクスとは，ソフトウェアをある側面から定量化した指標であり，ソフトウェアの特徴を把握する際に用いられる．ソフトウェアの保守性をより正確に把握できるようメトリクスの計測方法を改善することは，保守作業の効

率化に貢献する．

ソースコードの可読性の向上

修正すべき箇所の特定や修正方法の検討にあたり，作業者は既存のドキュメントやソースコードを直接読んでソフトウェアの挙動を理解する必要がある．特に機能面での追加や問題訂正を行う場合は，広範囲に渡ってソースコードを確認しなければならないこともあるが，保守を担当する作業者は，そのソフトウェアが最初に開発された時点の開発者とは異なることが多い [22]．ソースコードを読む作業は保守作業における開発者の行動の大半を占めるといわれているため [17, 23, 24]，ソースコードの可読性を向上することで，保守作業の効率化に貢献できる．

ソースコードに対する欠陥限局の精度向上

テスト工程では，実際に修正した箇所だけではなくソフトウェア全体に渡ってテストを行う必要がある．テストを実施した際，問題が見つかった場合は問題箇所の特定と修正，すなわちデバッグ作業が必要となる．デバッグもまた，多大な労力とコストを必要とする作業であり，デバッグ作業がソフトウェア開発コストの過半を占めるという報告もある [25, 26]．デバッグ支援の研究分野の一つに，ソースコードに含まれる欠陥箇所を推測する，欠陥限局という技術がある．欠陥限局の精度を向上することは，保守作業の効率化に繋がる．その手段としては，手法そのものを改善するというアプローチだけでなく，ソースコードを欠陥限局に適した記述とするアプローチが考えられる．なお，ソースコードが欠陥限局に適するかどうかの観点は，1.1.1 節に示した保守性の副特性のうち，解析性が示す「修正しなければならない部分を識別することが可能であることについての有効性及び効率性の度合い」に含まれると考えることができる．ただし，ここで「修正しなければならない部分を識別する」動作の主体となるのは保守作業者といった人ではなく，欠陥限局手法そのものであるため，人にとって識別しやすいソースコード記述と，欠陥限局手法にとって識別しやすい記述は異なる可能性がある．

1.2 研究の概要

前節の通り，メトリクス計測の改善，ソースコードの可読性の向上，及びソースコードに対する欠陥限局の精度向上は，ソフトウェア保守の効率化に向けて効果が期待できる．本研究では，上記の観点からソフトウェア保守を効率化することを目指し，ソースコードの内部構造に着目して保守性を計測または改善する次の三つの研究を行った．

1. ソースコード簡略化の前処理によるメトリクス計測方法の改善
2. プログラム文の並べ替えによるソースコードの可読性向上
3. ソースコードに対する欠陥限局の適合性計測

以降，各研究の背景と概要について述べる．

1.2.1 ソースコード簡略化の前処理によるメトリクス計測方法の改善

保守性を改善するためのリファクタリング作業として，理解性の低いモジュールを特定し，改善する活動が行われている．理解性の低いモジュールの特定に用いられるメトリクスとして，ソースコードの複雑度を表すメトリクス（以降，複雑度メトリクス）がある．代表的な複雑度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多いということは，規模が大きくそれだけ複雑さも増すといえる．したがって，コード行数は基本的な複雑度メトリクスの一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 [27]．プログラム制御の流れを有向グラフで表現する際，文や式をノードで表し，ノード間の遷移をエッジで表す．このとき，ノードの数を v，エッジの数を e とすると，サイクロマチック数は 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 [28]．クラスの重み・結合度・凝集度・応答数・継承の深さ・子クラスの数の計六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（不具合が顕在化しそうなモジュール）の検出に有効であるといわれている [29]．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 [30]．

なかでも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 [28,31–33]．たとえば，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 [29]．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 [31]．

サイクロマチック数がソースコードの理解性を正確に表すことができない要因の一つとして，ソースコード中の繰り返し構造の存在が挙げられる．Jbara らが Linux カーネルを対象に行った調査では，サイクロマチック数の非常に大きな関数の多くは，単純な if 文や switch 文中の case 文などが連

続して繰り返されることによって構成されており，そのような関数は人が見て複雑だと感じないことが報告されている [34]．すなわち，単純な構造の繰り返しが，サイクロマチック数とソースコード理解性の間に乖離が生じている要因の一つとなっているといえる．行数についても，単純な構造が繰り返されることによって行数が大きくなるため，繰り返し構造の存在により行数とソースコード理解性の間に乖離が生じる．

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発見する，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前処理を提案する．提案手法はソースコード中に存在する繰り返し構造を検知し，それらを折りたたむことでソースコードの簡略化を行う．これにより，サイクロマチック数や行数などのメトリクス値とソースコード理解性の間に生じた乖離を低減できる．提案手法を対象ソフトウェアに適用した後にメトリクス値を計測することで，人が複雑だと感じるモジュールの特定をより効率的に行うことができる．

本研究では，提案手法の適用後に計測されたメトリクス値がソースコードの理解性をより正確に表すことができるかどうかを評価するために，被験者の協力を得て評価実験を行った．その結果，提案手法を用いることで，被験者が複雑とみなしたモジュールをより効率的に特定できたことを確認した．

1.2.2 プログラム文の並べ替えによるソースコードの可読性向上

ソフトウェアの開発や保守作業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 Conventions [35] 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．たとえば，変数やメソッドなどの識別子は使用意図が分かるよう，長すぎず短すぎない命名を行うべきだと述べられている．また，メソッドの中身を読みやすくするための空行の位置や，制御構造などの階層構造を認識しやすくするためのインデント（字下げ）の方法についても言及されている．

ソースコード上の特徴と読みやすさに関する研究はこれまでに多く行われている．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 [31]．その結果，ソースコードの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．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可読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 [36]．その結果，最も広く用いられている基準は Java Code Conventions で定められている基準であったが，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．

ソースコードの可読性を向上するためのリファクタリング手法も多く提案されている．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た [37]．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 [38]．Tsantalis と Chatzigeorgiou は，ソースコード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 [39].

本研究では，プログラム文を並べ替えることで，ソースコードの可読性を向上させるリファクタリング手法を新たに提案する．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といわれているが [40]，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がある [41]．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べ替え操作が有効である．提案手法では，変数を内側のブロックに移動させてスコープを小さくする，およびブロック内での文の入れ替えにより定義と参照の間の距離を短くする，という二つの戦略に基づいてソースコード中の文の順番を入れ替える．提案手法を用いて，Java で記述されたオープンソースソフトウェアのメソッドに対して並べ替えを行い，被験者の協力を得て評価を行ったところ，提案した並べ替え手法によりメソッドの可読性が向上したとの結果を得た.

1.2.3 ソースコードに対する欠陥限局の適合性計測

ソースコード中の欠陥箇所を推測する手法の一つとして，実行経路情報を用いた欠陥限局（Spectrum-Based Fault Localization，以降，SBFL）に関する研究が盛んに行われている [42]. SBFL は，失敗するテストケースによって実行される文は欠陥を含む可能性が高いという考えに基づき，テストケースごとの成否と，ソースコード中で実行される文の情報（以降，実行経路情報）を用いて，欠陥を含む文を推測する技術である．SBFL 技術では，ソースコード中の各文に対して欠陥を含む可能性（以降，疑惑値）が数値化される．欠陥を含む文の疑惑値が他の文と比べてより高い値であるほど，SBFL 結果がより正確であるといえる.

高水準言語には様々な記述方法が用意されており，開発者は自身の好みや組織あるいはプロジェクトの方針にしたがって，必要な機能の実装方法を決定する．実装方法が変わるとテストケースごとの実行経路情報が変わる可能性があり，さらに文の疑惑値やその順位も変わる可能性がある．したがって，ソースコード自体が SBFL を用いた欠陥箇所の特定にどの程度適しているかという特性を持っていると考えることができる.

本研究では，SBFL の精度向上のアプローチとして，上記のようなソースコードの特性に着目し，あるソースコードに対する SBFL を用いた欠陥限局の結果がどの程度正確かを，ソースコードの SBFL 適合性として提案する．SBFL 適合性は，あるソースコードに対する SBFL 適用技術との親和性の一種であり，品質特性である保守性，およびその副特性である解析性に含まれる一つの観点とみなすことができる．ソフトウェアの品質特性の一つの観点として SBFL 適合性を扱うことにより，下記の活動が可能になる.

- 現在のソースコードに対する SBFL 結果がどの程度信頼できるかを事前に把握する．信頼できると判断された場合は，SBFL 技術を利用することにより欠陥限局を行い，その結果を利用して開発者がデバッグ作業を行えばよい.
- SBFL を利用した欠陥限局を行う前に，SBFL 適合性が低いソースコードに対し，SBFL 適合性が向上するよう一時的なソースコード変換を行う.

また，本研究では SBFL 適合性の一つの評価指標として，SBFL スコアの計測手法を提案する．具体的には，すべてのテストケースを通過するソースコードに対して，ミューテーションテスト [43] の技術を活用し，様々な箇所に意図的に欠陥を発生させ，SBFL を実行する．SBFL によってどの程度正確に欠陥箇所を特定できたかを計測することにより，元のソースコードの SBFL スコアを計測する．リファクタリングを題材に，ソースコード構造の違いによる SBFL スコアの計測結果を比較したところ，SBFL スコアを向上させるソースコード構造の変換例を発見した．

1.3 各章の構成

2 章ではソースコード簡略化の前処理によるメトリクス計測方法の改善について述べる．3 章ではプログラム文の並べ替えによるソースコードの可読性向上について述べる．4 章ではソースコードに対する欠陥限局の適合性計測について述べる．最後に，5 章で本研究のまとめと今後の研究の方向性について述べる．

第 2 章

ソースコード簡略化の前処理によるメトリクス計測方法の改善

2.1　導入

ソフトウェアの大規模化，複雑化に伴い，ソフトウェア保守を効率的に行うことは非常に難しくなっている．ソフトウェアライフサイクルのうち，保守作業が占める割合は非常に高いことが指摘されており [3]，ソフトウェア保守の効率化が求められている．ソフトウェアの保守を行うためには，保守対象となるソースコードを理解する必要がある．しかし，ソースコードを読み，理解することは，ソフトウェア保守の全工程の中で最もコストの高い作業であるといわれている [17]．したがって，理解性の低いモジュールを特定し，改善することによって，その後の保守作業を効率的に行うことができる．理解性の低いモジュールを特定するための方法として，ソフトウェアメトリクス（以降，メトリクス）を用いる方法が挙げられる．メトリクスとはソフトウェアの特性を定量的に知るための指標であり，これまでに様々なメトリクスが提案されている [27, 28]．理解性の低いモジュールの特定に用いられるメトリクスとして，ソースコードの複雑度を表すメトリクス（以降，複雑度メトリクス）がある．McCabe のサイクロマチック数 [27] は，モジュールの制御パス数を表す複雑度メトリクスであり，伝統的に用いられている複雑度メトリクスの一つである．しかし，サイクロマチック数は必ずしもソースコードの理解性の低さを表しているわけではない．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，サイクロマチック数とソースコード可読性との相関が低いことを報告している [31]．すなわち，理解性の低いモジュールを特定しようとする際，用いるメトリクスとしてサイクロマチック数は不向きな場合があるといえる．その改善を目的として，pmccabe [44] など，サイクロマチック数の計測方法を拡張したツールは幾つか提案されているが，それらを用いた評価実験は行われていない．ソースコードの行数も，理解性の低いモジュールの特定に用いられている．一般的に行数の大きいモジュールは理解性が低いと考えられており，そのようなモジュールはより小さい複数のモジュールに分割すべきであるとされている [18]．しかしながら，サイクロマチック数と同様に，行数も必ずしもソースコードの理解性を表しているとはいえない．

このように，既存のメトリクスがソースコードの理解性を正確に表すことができない要因の一つと

して，ソースコード中の繰り返し構造の存在が考えられる．Jbara らが Linux カーネルを対象に行った調査では，サイクロマチック数の非常に大きな関数の多くは単純な if 文や switch 文中の case 文などが連続して繰り返されることによって構成されており，そのような関数は人が見て複雑だと感じないことが報告されている [34]．すなわち，単純な構造の繰り返しが，サイクロマチック数とソースコード理解性の間に乖離が生じている要因の一つとなっているといえる．行数についても，単純な構造が繰り返されることによって行数が大きくなるため，繰り返し構造の存在により行数とソースコード理解性の間に乖離が生じる．

そこで本研究では，サイクロマチック数や行数を用いて理解性の低いモジュールを発見する，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前処理を提案する．提案手法はソースコード中に存在する繰り返し構造を検知し，それらを折りたたむことでソースコードの簡略化を行う．これにより，サイクロマチック数や行数などのメトリクス値とソースコード理解性の間に生じた乖離を低減できるため，提案手法を適用した後にメトリクス値を計測することで，複雑なモジュールの特定を効率的に行える．

本研究では，提案手法の適用後に計測されたメトリクス値がソースコードの理解性をより正確に表すことができるかどうかを評価するために，被験者の協力を得て評価実験を行った．その結果，提案手法を用いることで，被験者が複雑とみなしたモジュールをより効率的に特定できたことを確認した．また，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ソースコードからそれぞれメトリクス値を計測し，比較を行った．その結果，多くのソフトウェアに繰り返し構造が存在し，提案手法が有効に働く可能性の高い事例が多数実在することが確認できた．

2.2 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 (Abstract Syntax Tree，以降，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されたソースコードに対し，サイクロマチック数などのメトリクス値を計測すると，従来の値よりも低い値が計測される．これにより，単純な構造を持つモジュールに対するメトリクス値が小さくなるため，繰り返し構造を持たない複雑なモジュールに対するメトリクス値が相対的に高くなる．したがって，提案手法を前処理として適用することで，メトリクス値を用いた理解性の低いモジュールの特定をより効率的に行うことが可能となる．

```
  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

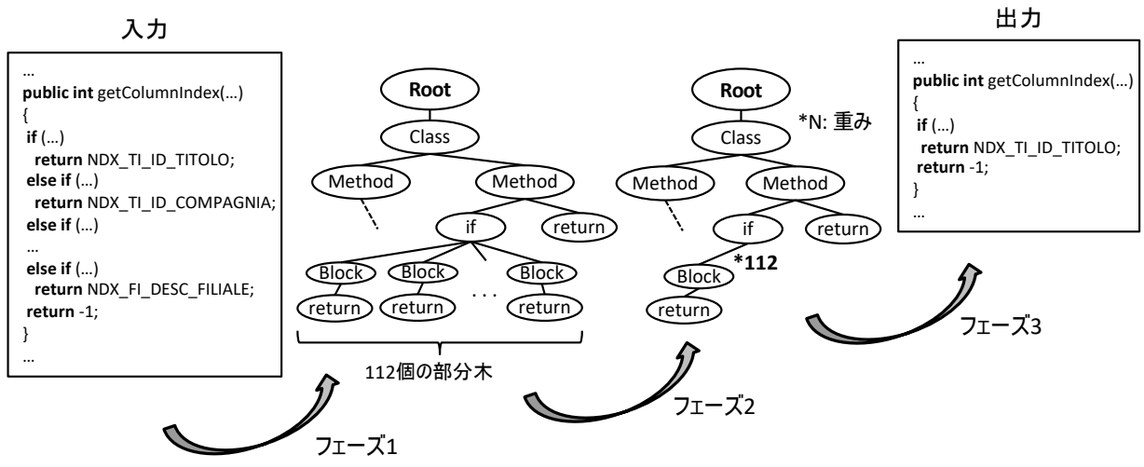


図 2.2　提案手法の概要

2.3　提案手法

本手法は，ソースコードの AST を入力として，以下の流れで実行される．

フェーズ 1：　入力されたソースコードから AST を構築する
フェーズ 2：　AST 中の繰り返し構造を折りたたむ
フェーズ 3：　折りたたまれた AST を元に，簡略化されたソースコードを出力する

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を図 2.2 に示す．

2.3.1　準備

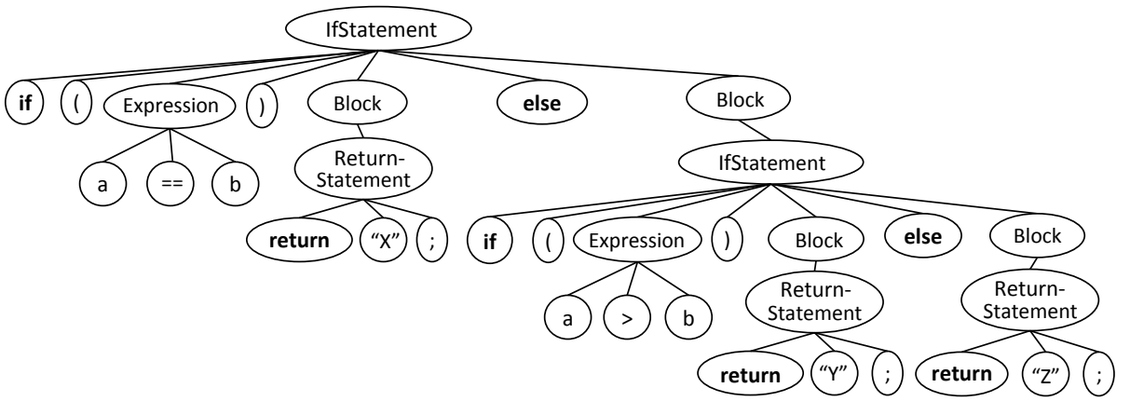
AST とは，ソースコードの構文情報を木構造で表したものである．AST の例を図 2.3 に示す．

AST 上のノードは構文上の一つの要素を表し，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 上の根にあたる *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，本研究で用いる 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 2.3(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．

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

(a) ソースコード



(b) AST

図 2.3　AST の例

前順走査 (preorder traversal)

まず根ノードを訪れる．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 2.3(b) に対して前順走査すると，図 2.3(a) の二つの 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．最後に根ノードに訪れる．例えば図 2.3(b) に対して後順走査すると，図 2.3(a) の二つの if 文は，その出現順とは逆順に訪れられる．

2.3.2　フェーズ 1：AST の構築

まず与えられたソースコードを読み込んで，AST を構築する．ただし，図 2.1(b) のように if 文の後に連なる else-if 節については，AST の変形処理を同時に行う．変形処理とは，AST 上で else 節を表すノードの直下に 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．この処理の流れを図 2.4 に示す．変形処理を行うことで，後に述べるフェーズ 2 において，連続する else-if 節を繰り返し構造であるとみなすことがで

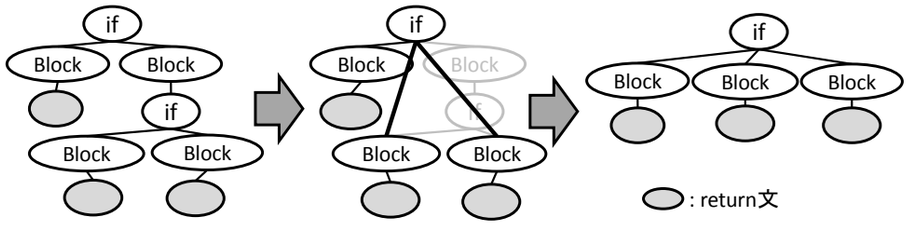


図 2.4　else-if 節の変形

きる．また，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．そのため，本論文では AST にそれらの情報を表示していない．

2.3.3　フェーズ 2：繰り返し構造の折りたたみ

本研究における繰り返し構造を以下に定義する．

定義 2.3.1（繰り返し構造）　AST 上の兄弟ノードはソースコード上の出現順による順序性をもっている．AST 上で連続する兄弟ノードが互いに類似しているとき，それらの兄弟ノードからなる部分木の集合を繰り返し構造と定義する．

ノードの類似基準を表 2.1 に示す．また，Java を対象とした場合の文の種類を表 2.2 に示す．式文については，式文として頻繁に出現するメソッド呼び出し式と代入式のみをそれぞれ独立した種類の文として扱い，増分式，減分式，クラスインスタンス生成式は，その他の式文としてまとめている．

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
共に子を持たない場合	文の種類と重みが同じ.
共に子を持つ場合	ノードを根とする部分木が同形かつ 対応する全ての子ノードの対が類似. (子ノードの対に表 2.1 を再帰的に適用)

表 2.2　文の種類（Java）

カテゴリ	文の種類
ブロック文	if 文，for 文，拡張 for 文，while 文，do-while 文，switch 文， case 文（default 句を含む），try 文，catch 節，finally 節，synchronized 文.
単文	式文，変数宣言文，assert 文，break 文，continue 文， return 文，throw 文，空文.
式文	メソッド呼び出し文，代入文，その他の式文.

更に，表 2.1 中の「ノードの重み」を以下に定義する．

定義 2.3.2（ノードの重み） あるノードを根とする部分木が繰り返し構造を構成する単位となっているとき，その繰り返し構造中の繰り返し回数をそのノードの重みと定義する．

表 2.1 の基準によって繰り返し構造とみなされる例を図 2.5(a)，2.6(a) に挙げる．これらの図におけるノードの模様はノードを構成する文の種類を表しており，同じ模様を持つノードはそれらのノードを構成する文の種類が同じであることを示している．図 2.5(a) は子を持たないノードによって構成される繰り返し要素の例を表しており，三つのノードが一つの繰り返し構造を構成している．一方，図 2.6(a) に示す例は互いに子を持つノードによって構成される繰り返し構造の例であり，三つのノードからなる部分木を繰り返しの単位とする，繰り返し回数 2 の繰り返し構造が存在する．繰り返し単位となる部分木の根ノードの対に表 2.1 を適用した場合，これらのノードは共に子を持つため，表下段の処理が実行される．この場合は，部分木はいずれも二つの子ノードを持っていることから同じ形状と判定されるため，部分木を構成する全てのノード対が類似しているかを判定するために，根ノードのそれぞれの子ノードに対して再帰的に表 2.1 が適用される．

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．例えば図 2.9 に示すソースコードは，代入文とメソッド呼び出し文という二種類の文による記述の繰り返しである．このような記述を繰り返し構造と判断できるよう，複数種類の兄弟ノード列が AST 上で連続している場合も繰り返し構造とみなす．このような繰り返し構造の例を図 2.7(a) に挙げる．この例の場合，二つのノードからなるノード列を繰り返し単位とする繰り返し回数 2 の繰り返し構造が存在する．

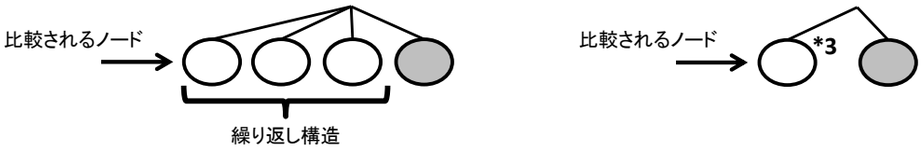
続いて，本研究における繰り返し構造の折りたたみの定義を以下に示す．

定義 2.3.3（繰り返し構造の折りたたみ） 単位となる部分木 (列) を一つだけ残して削除し，繰り返し回数を表す重みを根ノードに対してもたせる処理のことを，繰り返し構造の折りたたみと定義する．

繰り返し構造の中には入れ子になったものも存在する．例えば図 2.10 に示すソースコードには，メソッド呼び出し文の繰り返しと，それを内部に含む if 文の繰り返しが存在する．このような繰り返し構造の例を図 2.8(a) に示す．この場合，メソッド呼び出し文を先に折りたためるよう，AST 上の葉に近いノードほど先に折りたたみを行う．

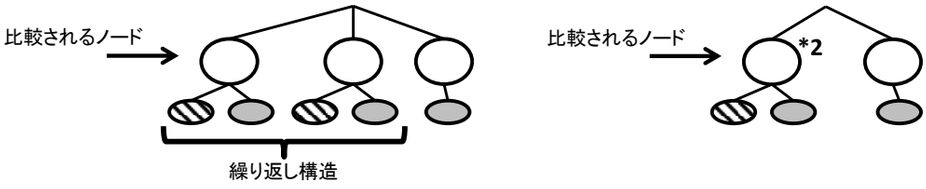
これまでに述べた繰り返し構造に対して折りたたみ処理を行う例を，図 2.5〜2.8 に示す．

以上を踏まえて，フェーズ 2 は次の手順で実行される．まず，AST を後順走査して兄弟ノードを取得する．続いて，兄弟ノード中の繰り返し構造を特定する．このとき，繰り返された文の単位が小さいものから順に判断し，兄弟ノード中に繰り返し構造が見つからなくなるまで再帰的に折りたたみを行う．図 2.10 のソースコードに対してフェーズ 2 を適用したものを図 2.11 に示す．なお，図 2.11 中の $*2$ というラベルは，そのラベルが付いているノードの重みが 2 であることを表している．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文献 [45] において述べられている．本節で紹介したフェーズ 2 を用いると，文献 [45] 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．



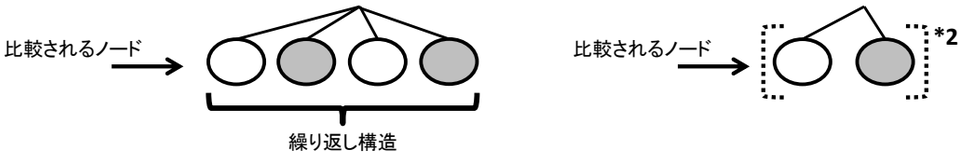
(a) 元の AST　　(b) 折りたたみ後の AST

図 2.5　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 2.6　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 2.7　複数のノード（ノード列）による繰り返し構造

(a) 元の AST　　(b) 折りたたみ後の AST

図 2.8　繰り返し構造を要素に持つ繰り返し構造

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

図 2.9　二種類の文による記述の繰り返し構造

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

図 2.10　入れ子になった繰り返し構造

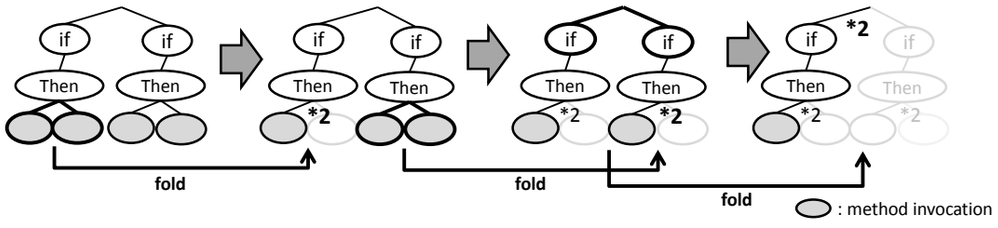


図 2.11　AST を折りたたむ例

2.3.4　フェーズ 3：ソースコード生成

最後に，折りたたまれた AST から簡略化されたソースコードを生成する．

なお，本提案手法ではソースコード中の繰り返し構造の一部を削除しているため，提案手法適用後のソースコードはコンパイルエラーを含んでいる可能性がある．本手法はサイクロマチック数や行数を計測するためだけのソースコード簡略化であるため，それらのメトリクスが計測できればコンパイルエラーが含まれていても問題は無い．ただし，エラーが含まれているソースコードを計測できない仕様のメトリクス計測ツールを用いる場合は，正常に計測できなくなる可能性について留意する必要がある．

2.4　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 [46] 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 2.3 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，次の二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はソースコードの理解性と関係があるか．
実験 2：　計測されるメトリクス値が変化する事例がどの程度存在するか．

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．

2.4.1　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

表 2.3　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

- while 文
- do-while 文
- switch 文中の case 文
- 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研究では表 2.4 に記載の名前で表現する．

2.4.2 実験 1：ソースコードの理解性とメトリクス値の比較

実験 1 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．

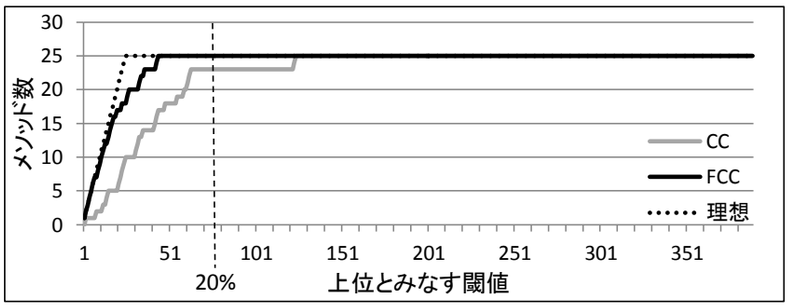
1. 対象ソフトウェア中のメソッド一つ一つについて，理解しづらいかどうかの評価を被験者に行ってもらう．
2. 被験者によって理解しづらいと評価されたメソッドを**正解メソッド**とし，そのメソッドの集合を**正解集合**とする（正解集合は被験者の数だけ得られる）．
3. 全てのメソッドをそれぞれのメトリクス値の降順に並べ，上位に含まれる正解メソッドの数を比較する．

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．なお，本実験では被験者がメソッドの処理を正確に理解したかどうかの判定は行わず，被験者自身が感じた理解のしづらさという主観的な評価のみを用いている．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．

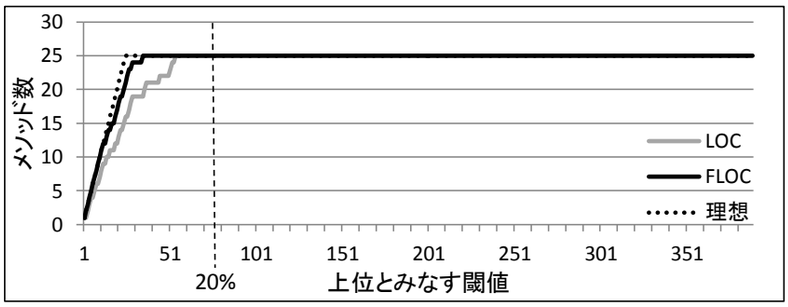
まず，上位とみなす閾値を変化させて上位に含まれる正解メソッド数を計測した．図 2.12 は，各メトリクス値上位に含まれる被験者 B の正解メソッド数を，上位とみなす閾値ごとに表したグラフである．正解メソッドのみを上位に含む場合が理想であるため，各メトリクスによる正解メソッド数の推移がグラフ中の点線で示した形に近いほど，被験者によるソースコード理解性と関係があることを示している．グラフより，CC よりも FCC の方が，また LOC よりも FLOC の方が，被験者 B に

表 2.4　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



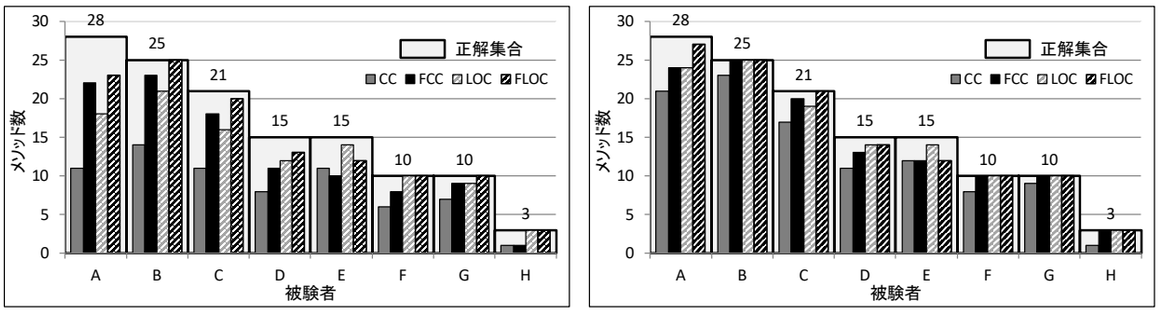
(b) 行数

図 2.12　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)

とって理解しづらいメソッドを見つけた割合が高くなっている．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．

更に，それぞれの被験者について，メトリクス値上位のメソッドから順にメソッドを調査したとき，全ての正解メソッドを発見するために必要となるメソッド数を調査した．その結果を表 2.5 に示す．各メトリクス名の列にある数値が低いほど，より少ないメソッド数を調査することで全ての正解メソッドを特定することができるということを意味しており，差分の列にある数値が高いほど，提案手法の適用による効果が大きいことを意味している．表 2.5 より，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，行数の場合は 8 人中 5 人の被験者について，提案手法の適用によってより効率的に理解性の低いメソッドを特定することができるといえる．

次に，メトリクス値の上位 10% と上位 20% に含まれる，各被験者に対する正解メソッドの数を図 2.13 に示す．例えば図 2.12 の例に示した被験者 B の場合，正解メソッドの数は 25 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．被験者 B の場合，上位 20% の比較では，CC 以外のメトリクス値上位には全ての正解メソッドを含んでいるが，上位 10% の比較では，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 B の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．



(a) 上位 10%　　(b) 上位 20%

図 2.13　メトリクス値上位に含まれる正解メソッドの数

このグラフに示した値を元に，各被験者の実験結果について以下の二つの値を算出した．

適合率：　メトリクス値上位に含まれるメソッドのうち，正解メソッドが占める割合．

再現率：　全ての正解メソッドのうち，メトリクス値上位に出現する正解メソッドの割合．

各メトリクス値上位に含まれるメソッド数に差がある例として，被験者 A の実験結果に対する適合

表 2.5　全ての正解メソッドの発見に必要なメトリクス値上位のメソッド数

被験者	正解メソッド数	CC	FCC	差分（CC-FCC）	LOC	FLOC	差分（LOC-FLOC）
A	28	161	147	14	138	120	18
B	25	124	44	80	56	35	21
C	21	124	121	3	96	53	43
D	15	124	44	80	140	144	-4
E	15	158	144	14	93	176	83
F	10	161	199	38	27	22	5
G	10	123	43	80	67	31	36
H	3	124	46	78	5	5	0

表 2.6　適合率・再現率（被験者 A）

		CC	FCC	LOC	FLOC
上位 10%	適合率	29%	58%	47%	61%
	再現率	39%	79%	64%	82%
上位 20%	適合率	27%	31%	31%	35%
	再現率	75%	86%	86%	96%

率および再現率の算出結果を表 2.6 に示す．表 2.6 より，被験者 A については，サイクロマチック数，行数のいずれについても，適合率，再現率ともに提案手法適用後の値の方が適用前よりも高いという結果が得られた．特に，上位 10% におけるサイクロマチック数に対する適合率，再現率は，提案手法の適用により大きく上昇していることが分かる．ここで，被験者 A の場合は正解メソッドが 28 であるため，上位 20% における適合率は最大でも 36% である．このため，上位 20% における適合率は再現率と比較して小さな値となっている．全被験者の結果を見ると，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，提案手法の適用により上位 20% における適合率および再現率が上昇した．一方，行数の場合，上位 20% において適合率および再現率が上昇したのは 8 人中被験者 A, C の 2 人のみであった．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．

上位 10% における実験結果では，サイクロマチック数については 8 人中 6 人の被験者において，行数については 8 人中 5 人の被験者において，提案手法の適用により適合率および再現率の上昇が確認された．この結果から，提案手法を用いることでより多くの理解しづらいメソッドがメトリクス値上位に現れていることが分かる．

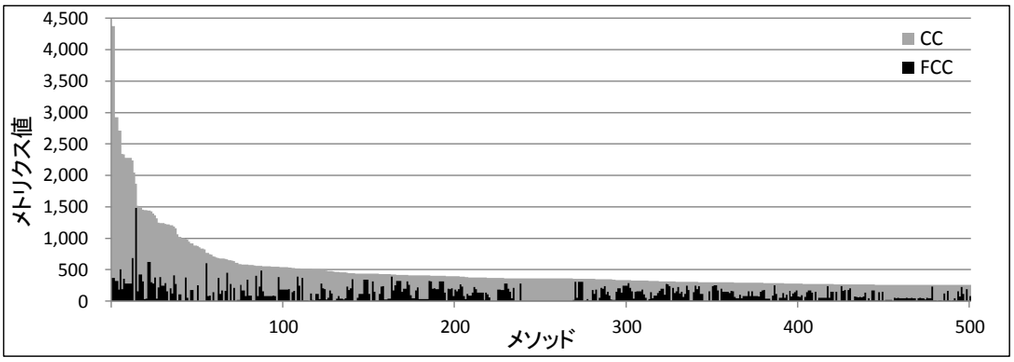
続いて，提案手法適用前後のメトリクス値上位に含まれる正解メソッドがどの程度重複しているかを調査した．その結果，ほぼ全ての正解メソッドは，提案手法適用前のメトリクス値上位に含まれていれば，提案手法適用後のメトリクス値上位にも含まれていることを確認した．例外として，被験者 E だけが理解しづらいとみなした 2 つのメソッドは，提案手法適用前のメトリクス値上位にのみ存在し，提案手法適用後のメトリクス上位には含まれなかった．このメソッドについての考察は後述する．

また，提案手法の適用により新たにメトリクス値上位に出現したメソッドに対する分析を行った．その結果, そのようなメソッドの中には，全ての被験者が理解しづらいと判断したメソッドが存在することを確認した．このメソッドは GUI コンポーネントの初期化を行うメソッドであり，提案手法を適用した場合はサイクロマチック数の上位 20% に出現したが，提案手法を適用しなかった場合は，サイクロマチック数の高い他のメソッドが上位に存在するため，発見することは容易ではなかった．すなわち，多くの人に理解しづらいとみなされるメソッドを，提案手法を適用することで効率的に特定することができたといえる．

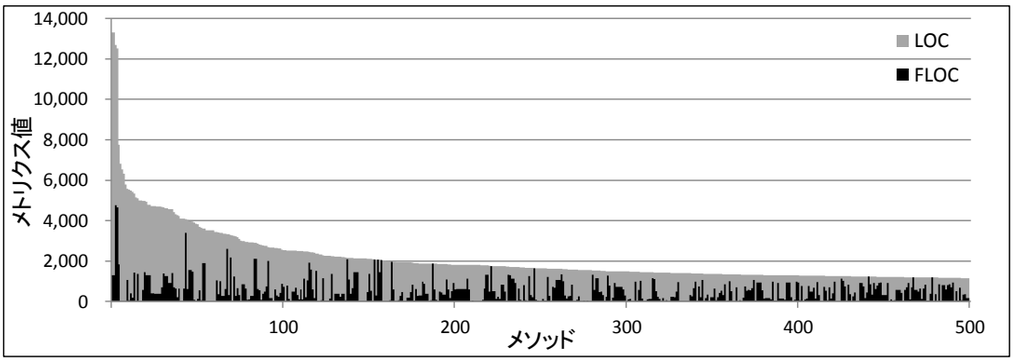
2.4.3 実験 2：計測されるメトリクス値の比較

実験 2 では，UCI source code data sets に含まれる約 13,000 のソフトウェア全てを対象として，提案手法適用の有無によってメトリクス値が変化するメソッドがどの程度存在するのかを調査する．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 2.14 に示す．二つのグラフともに，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．図 2.15 に提案手法適用前

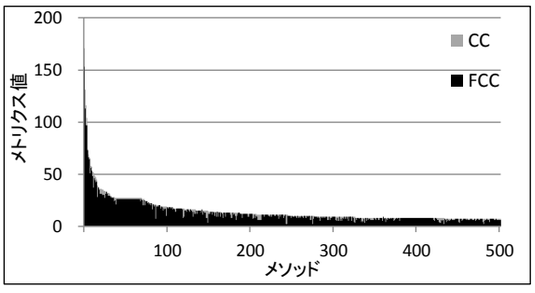


(a) サイクロマチック数

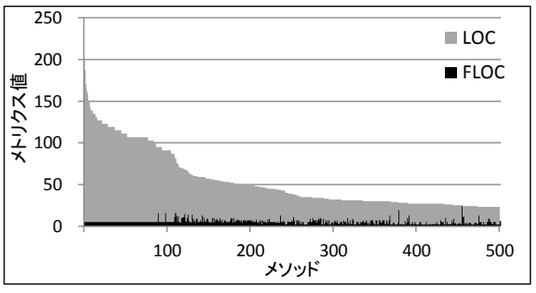


(b) 行数

図 2.14　メトリクス値の違い



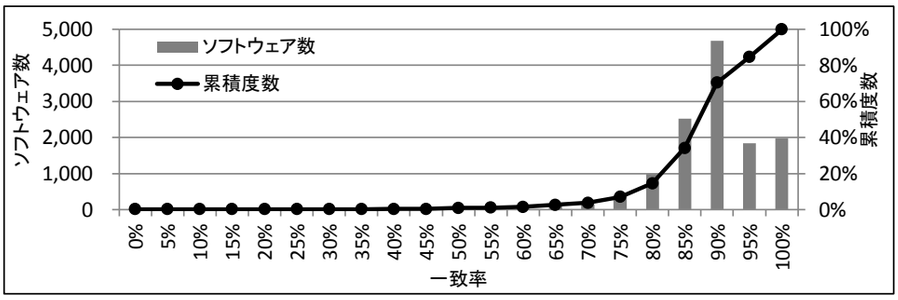
(a) 違いの少ない例



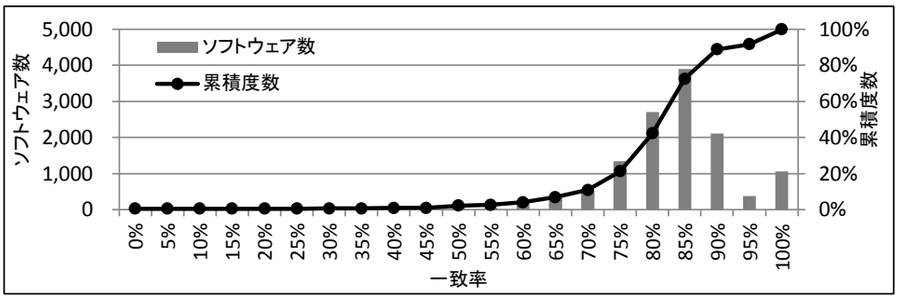
(b) 違いの多い例

図 2.15　各ソフトウェアにおけるサイクロマチック数の違い

後でサイクロマチック数が変化したメソッドが多数存在するソフトウェアと，その反対にサイクロマチック数の変化がほとんど見られなかったソフトウェアについて，サイクロマチック数上位 500 のメソッドにおける計測値の変化の様子を示す．このように，計測値にほとんど違いがないソフトウェアもあれば，計測値が大きく異なるソフトウェアもあり，ソフトウェアによって傾向が異なっていた．



(a) サイクロマチック数



(b) 行数

図 2.16　メトリクス値上位 20% における一致率

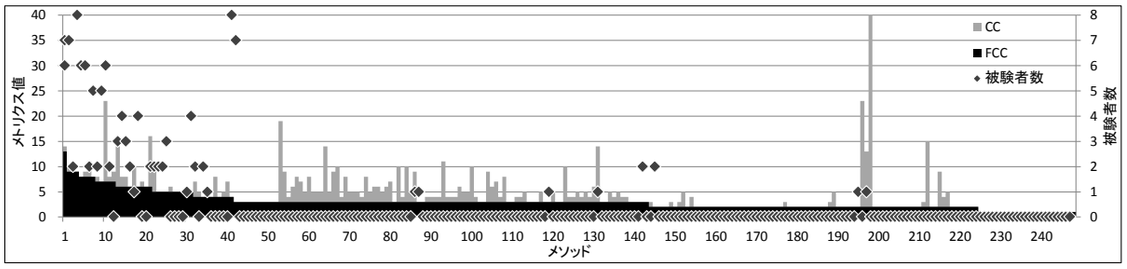
そこで，上記のような傾向をソフトウェアごとに得るため，各メトリクス値による上位 n 個のメソッド中に同じメソッドが含まれる割合として**一致率**を定義し，ソフトウェアごとに計測した．サイクロマチック数の場合，一致率は以下のように定義される．

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

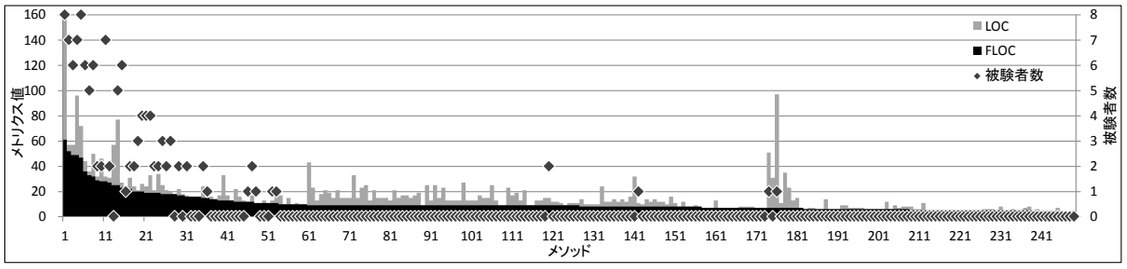
n は上位とみなすメソッドの数を表し，$CC(n)$ は CC の値による順位付け上位 n 個の集合を，$FCC(n)$ は FCC の値による順位付け上位 n 個の集合を表す．

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 2.16 に示す．図 2.16(a) は CC と FCC から，図 2.16(b) は LOC と FLOC から算出された一致率をそれぞれ表している．このグラフの x 軸は一致率を表しており，棒グラフは該当するソフトウェアの数を，折れ線グラフは全てのソフトウェアの合計を 100 としたときの累積度数をそれぞれ示している．例えば x 軸が 80% となっている箇所は，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



(a) サイクロマチック数



(b) 行数

図 2.17　メソッドのメトリクス値と正解とした被験者数の関係

ソッドが変化するという結果を得られた．

2.5　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．

2.5.1　実験 1 について

実験 1 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 2.17 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．またそれぞれの棒グラフは提案手法適用前後のメトリクス値 (左軸) を表しており，各点はそれぞれそのメソッドを理解しづらいと判断した被験者の数 (右軸) を表している．この結果からは，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．また，提案手法の適用によりメトリクス値が小さくなったメソッドは，そのメソッドを理解しづらいと判断した被験者数が少ないことが分かる．すなわち，提案手法の適用によって，従来のメトリクス値は高いが理解性が低いとはいえないメソッドのメトリクス値が小さくなっている．したがって提案手法の適用によってメトリクス値と理解性との関係がより強いものになったといえる．

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二つで，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．例えば図 2.18(a) のメソッドは複数の switch 文を含んでおり，各 case 文は単純な記述であったが，各 switch 文中の case 文の数や，case 文が内包する文の数が異なっていたことで，switch 文自体を全く折りたたむことができなかった例である．このメソッドの折りたたみ後の AST を図 2.18(b) に示す．case 文からなる部分木は同じ形に見えるが，繰り返し回数による重みが異なっているため，これ以上折りたたむことができない．

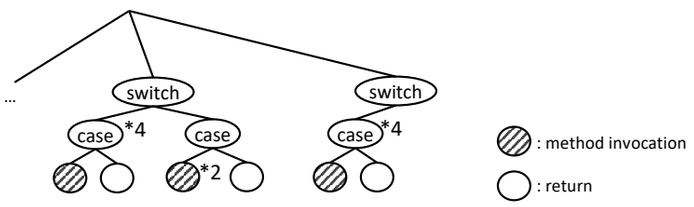
提案手法では，同じ構造を内包する文が連続していても，その繰り返し回数が異なる場合は類似した構造とみなしていないため，この例のような case 文の繰り返し回数が異なる switch 文を折りたたむことができなかった．しかし，どの被験者からも選ばれなかったということは，類似した文の繰り返し回数の違いは，人が理解する上で重要ではない可能性がある．繰り返し回数による違いを吸収した折りたたみを行うと，このようなメソッドのメトリクス値を低く計測することができるため，今後はそのような改善が必要であると考えられる．なお，このような false-positive に相当するメソッド数が多い場合，被験者にとって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

2.4.2 節で述べた，被験者 E だけが正解メソッドとして選択した 2 つのメソッドのような，提案手法適用によってメトリクス値上位に含まれなくなったメソッドが該当する．こ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であったが，単純な繰り返し構造を持っていたため，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っていた．被験者 E はこのようなメソッドを理解しづらいと判断していたため，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．

```
private String getSpecialModeDescription() throws ... {
  ...
  switch (values[1]) {
    case 0:
      desc.append("Unknown sequence number");
      break;
    ...
    default:
      desc.append(values[1]);
      desc.append("th in a sequence");
      break;
  }
  switch (values[2]) {
    case 1:
      desc.append("Left to right panorama direction");
      break;
    ...
  }
  ...
}
```

(a) 折りたたみ前のソースコード



(b) 折りたたみ後の AST

図 2.18　メトリクス値が高い非正解メソッド

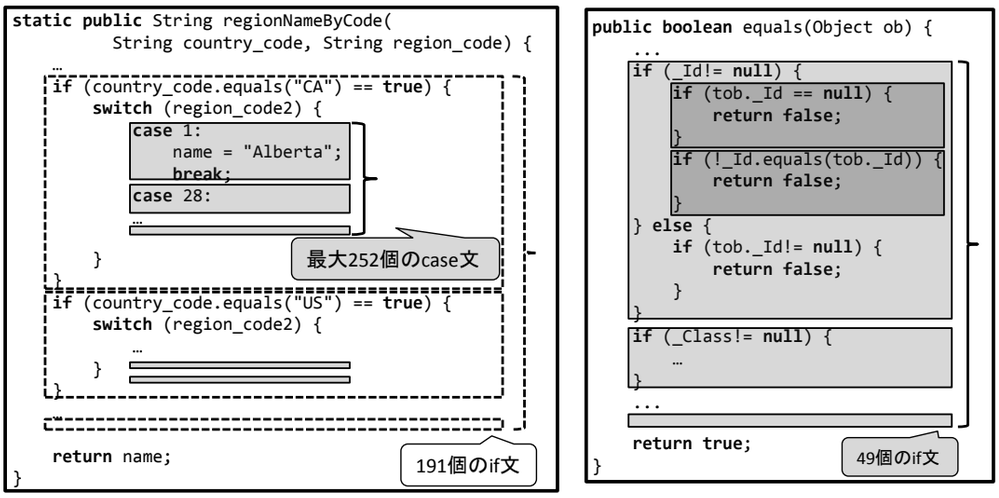
メトリクス値が低い非正解メソッド

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．

2.5.2　実験 2 について

まず，図 2.14(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 2.19(a) に示すように，一つの if 文の中に一つの switch 文を含む構造が 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巨大な構造であった．図 2.18 の例で述べたとおり，連続する switch 文に含まれるそれぞれの case 文の数が異なる場合は，提案手法によって，それらの case 文を内包する switch 文同士を類似した構造とみなすことができない．そのため，switch 文と if 文を折りたたむことができなかった．

続いて，図 2.15(b) に含まれるメソッドについて調査を行った．このソフトウェアに含まれる CC



(a) 全メソッド中 CC 値が最大のメソッド

(b) 図 2.15(b) に含まれるメソッドのうち，CC 値が最大のメソッド

図 2.19　発見された繰り返し構造

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 2.19(b) に示すようにネスト内に if 文を三つ含む 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 2.19(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理解しづらいメソッドの特定が容易になったといえる．

また実験 2 の結果から，提案手法の適用によってメトリクス値が変化するメソッドが多数存在することが確認された．この結果と実験 1 で得られた結果を勘案すると，提案手法が有効に働く事例が多数実在する可能性が高いと考えられる．

2.5.3　先行研究との比較

既存研究の中に，コードクローン検出における前処理としてソースコードの繰り返し部分を折りたたむ手法が提案されている [47, 48]．この既存手法はコードクローン検出に要する時間的，空間的コストの増大を抑制するために，簡単なソースコード解析のみによって折りたたみの処理を実現している．このため既存手法では，類似した処理を行っているソースコードであっても，そのソースコードを構成する字句の列に差異が含まれる場合，繰り返し部分として検知することができない．図 2.20 に示すメソッドは UCI data sets に含まれているソフトウェア中に存在するメソッドであり，同じような処理を行う try 文が多数繰り返されている．このメソッドに対して既存の折りたたみ手法を適用

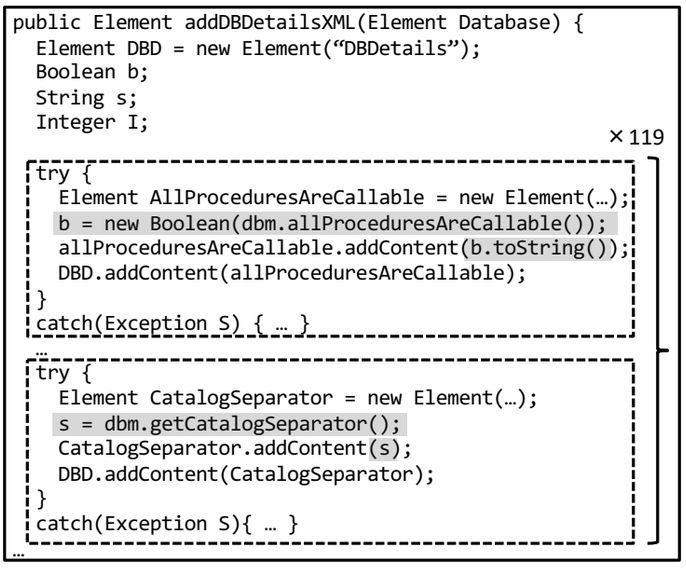


図 2.20　差異を含む繰り返し構造

した場合，try 文中の字句列に差異が存在するため，適切に折りたたみを行うことができない．これに対し，提案手法はメソッドの文構造のみに着目しているため，これらの try 文を繰り返し構造として検出し，折りたたみによって簡略化することができた．

2.6　妥当性について留意すべき点

本提案手法では AST を用いて繰り返し構造を判定する際，次の二つの観点で再帰的な処理が発生する．一つは，共に子を持つ部分木を比較する際，部分木が同形であれば部分木を構成する全ての子ノードに対して再帰的に類似判定を行う．もう一つは，兄弟ノード列を比較する際，比較するノード数を 1 から順に増やしながら再帰的に類似判定を行う．そのため，木構造が深く，かつ同形，すなわち同じ深さのネストに同じ数の文が存在する場合は，再帰的に類似判定を行うノード数が増えるため，処理時間が増大する恐れがある．

実験 1 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一つを対象にした．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，提案手法についてより詳細な評価を得ることができると考えられる．

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，今回の実験における被験者はある程度のソースコード読解力を有している可能性がある．すなわち，実験の被験者がソースコード読解力の高い者に偏っている可能性がある．また，被験者数が 8 人のみであることも，結果の偏りに繋

がっている可能性がある．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．

また，被験者が複雑とみなしたメソッドは，提案手法適用後の行数の上位に登場するものもあれば，提案手法適用後のサイクロマチック数の上位に登場するものもあった．本研究では，提案手法適用後に計測されるそれぞれのメトリクスが，オリジナルよりも理解性と関係があることを示したが，どちらのメトリクスが，どのような場合に優先されるかといった両メトリクスの比較評価は行えていない．

2.7 まとめと今後の課題

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．被験者実験の結果，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．更に，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．

今後の課題は以下の三点である．

- 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．
- 理解性の低いモジュール特定に効率的なメトリクスについて分析を行う．

第 3 章

プログラム文の並べ替えによるソースコードの可読性向上

3.1　導入

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 [3]．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 [17, 23, 24]．

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，プログラム理解に関する研究がこれまでに多く行われている．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 [31]．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 Conventions などが存在する [35]．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一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．また，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 [37]．

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 [18]．

本研究では，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を新たに提案する．

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 [31]．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 [41]．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べ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 [37, 49]．しかし，文の並べ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，文を自動で並べ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，変数の定義と参照間の距離に着目して，モジュール内の文を並べ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べ替えの行われたメソッドは可読性が向上したという結果が得られた．

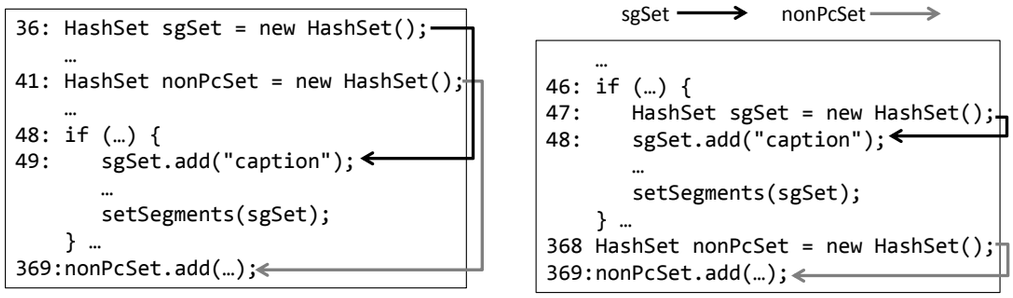
本研究の主な貢献を以下に記す．

- プログラムの振る舞いを変えることなく，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を提案した．
- 提案手法では，変数を内側のブロックに移動させてスコープを小さくする，およびブロック内での文の入れ替えにより定義と参照の間の距離を短くする，という二つの戦略に基づいてソースコード中の文の順番を入れ替えた．
- ブロック内での文の入れ替えについては，ブロック内に存在する全ての変数における，定義と参照の間の距離の総和を，可読性の指標値として活用した．
- 提案手法を用いて，Java で記述されたオープンソースソフトウェアのメソッドに対して並べ替えを行った．並べ替えの結果は，44 名の被験者によって評価された．その結果，実験対象とした 20 のメソッドのうち，16 のメソッドにおいて，提案した並べ替え手法によりメソッドの可読性が向上したとの結果を得た．
- 提案手法を用いても可読性が向上しなかったメソッドについて，その原因を考察した．また，考察を元にして，有用と考えられるリファクタリング支援環境の構想について述べた．

3.2 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，定義と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．これら二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．この移動を行うことによって，二つの変数の定義と参照の間の距離が縮まり，ソースコードの可読性が増すと考えられる．

本研究では，ソースコードの可読性を向上させるために，文の並べ替えを行う手法を提案する．



(a) 移動前　　(b) 移動後

図 3.1　プログラム文の移動例

図 3.1 の例に基づいて，具体的に次の二通りの移動戦略を用いる．

戦略 A　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 B　変数の定義と参照の間の距離を短くするため，共通のブロック内で文を移動する．

3.3　提案手法

3.3.1　変数の定義と参照間の距離の取得

本手法では，変数の定義と参照の関係を，データフロー解析を用いて得られる Def-Use chain（以下，DU チェイン）から取得する [50]．DU チェインとは，ある変数の定義と参照関係を表したものである．二つの文 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 DU チェインが存在する．

- 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 の再定義のない一つ以上の実行パスが存在する．

ただし，「文 s は変数 v を定義する」とは，以下のいずれかの動作を表す．

- 文 s において，変数 v に対して代入処理が行われている．
- 文 s において，変数 v が指すオブジェクトの状態が変更されている．

変数の定義と参照の間の距離は，DU チェインを用いて算出する．ある DU チェイン c の距離 $distance(c)$ を，DU チェイン c によって結ばれた二つの文の間に存在する文の数とする．このとき，あるソースコード中のあるブロック b に含まれる全ての DU チェインの総距離 $totaldistance(b)$ は，b に含まれる全ての DU チェインの集合 $DUchain(b)$ を用いて以下の式で表す．なお，「DU チェイン c

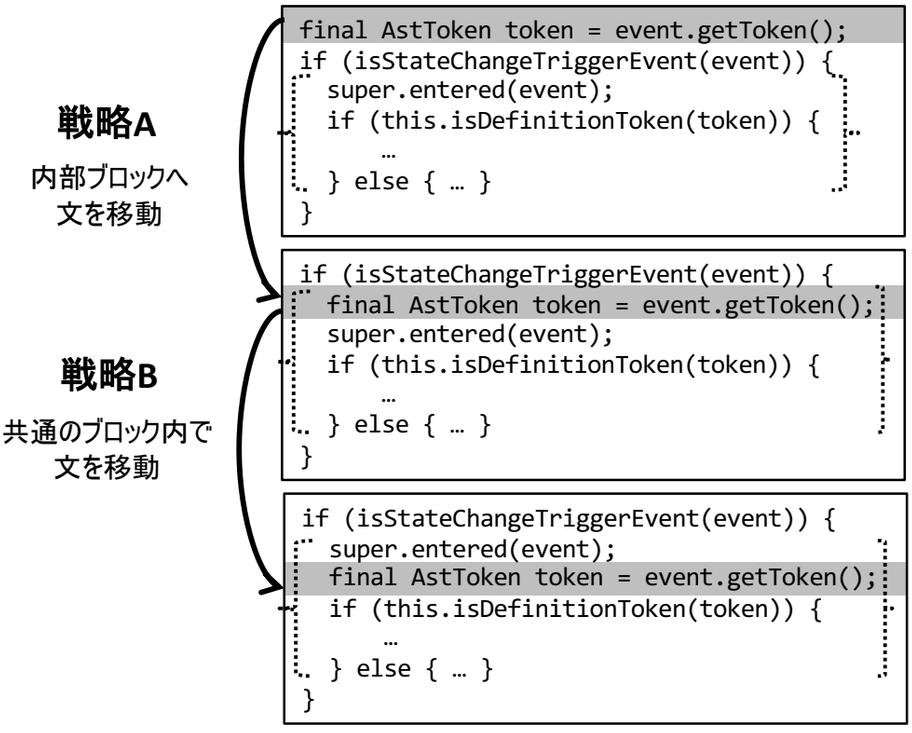


図 3.2　フェーズ 2 の適用例

がブロック b に含まれる」とは，c を構成する二つの文がともに b 内に存在することを表す．

$$totaldistance(b) = \sum_{c \in DUchain(b)} distance(c) \tag{3.1}$$

3.3.2　手法の概要

本手法は，ソースコードを入力として以下の流れで文の並べ替えを行う．

フェーズ 1.　ソースコードから抽象構文木（AST）を構築する．

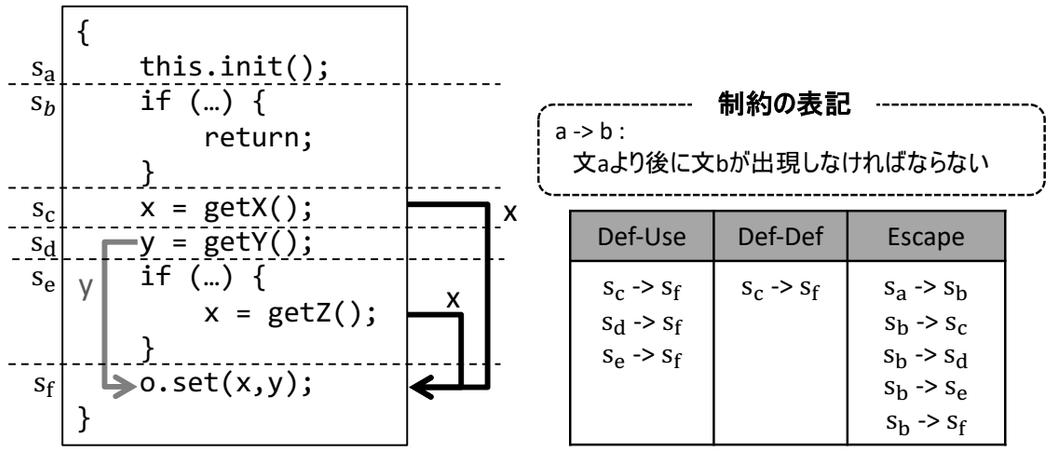
フェーズ 2.　AST を後順走査し，訪れたブロックに対して次の二つの移動戦略を適用する．

戦略 A　内部ブロックへ文を移動する．

戦略 B　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，二つの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 $totaldistance$ が最小になるよう，ブロック内の文を並べ替える．



(a) 並べ替え対象の文　　(b) 図 3.3(a) に対する順序制約

図 3.3　文の順序制約の例

3.3.3　戦略 A の実現方法

以下の全て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する．

- 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 DU チェインが存在する．
- 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およびそれが指すオブジェクトは，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．

3.3.4　戦略 B の実現方法

着目中のブロック内の文を並べ替える際，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 $totaldistance$ が最小のものを選択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 B を適用する場合を考える．このとき，並べ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，並べ替え対象の文自体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．

プログラムの振る舞いを変えない範囲で文の並べ替えを行うために，提案手法では，プログラム文が満たさなければならない順序制約として次の三つを定める．

Def-Use 制約

ある変数の定義と参照の関係にある二つの文は，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 で定義された変数 y を参照しており，この二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，並べ替え対象の文が，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．この二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をまたいで文を移動することはできない．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．よって，文 s_a を文 s_b よりも後に移動することも，文 s_c，s_d，s_e，s_f を文 s_b の前に移動することもできない．なお，ブロック外へのジャンプ命令とは，return 文の他に continue 文，break 文や，Java 言語の場合は throw 文，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は下記の手順で行われる．

手順 1　順序制約を満たす全ての順列（文の並び）を生成する．
手順 2　手順 1 で生成した順列の中から $totaldistance(b)$ が最小である順列を抽出する．

しかし，$totaldistance(b)$ 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．

手順 3　抽出した順列の中で，オリジナルの順列（入力メソッドにおける文の並び）と最も近いものを一つだけ抽出する．

本研究では，変数の定義と参照の間の距離にのみ着目した文の並べ替えを行うため，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う．この処理では，オリジナル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，二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，手順 2 で得られた全ての候補とオリジナルの順列との間で 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．

3.4 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べ替えを行うことによって，ソースコードの可読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．

3.4.1 準備

TVBrowser は約 14 万行であり，約 3,700 のメソッドが含まれていた．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べ替えが行われた．また，実行時間は約 56 分であった．本実験では，文の並べ替えが行われたメソッドのうち，20 個を用いてアンケート調査を行った．このアンケートでは，各被験者は 20 個のメソッドに対して，オリジナルのソースコードと並べ替え後のソースコードを比較し，どちらの方が可読性が高いかを判定した．アンケートの準備は以下の手順で行った．

手順 1　20 個の対象メソッドについて，オリジナルのソースコードと並べ替え後のソースコードからコメントおよび空行を取り除く．また，オリジナルと並べ替え後のソースコード間でインデントや改行位置などのフォーマットを統一する．

手順 2　各メソッドにつき，オリジナルと並べ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，その結果を表 3.1，表 3.2 に示す．

3.4.2 結果

実験結果を図 3.4 に示す．グラフ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，有意水準 1% で差がある（p 値は 0.002 であった）こと

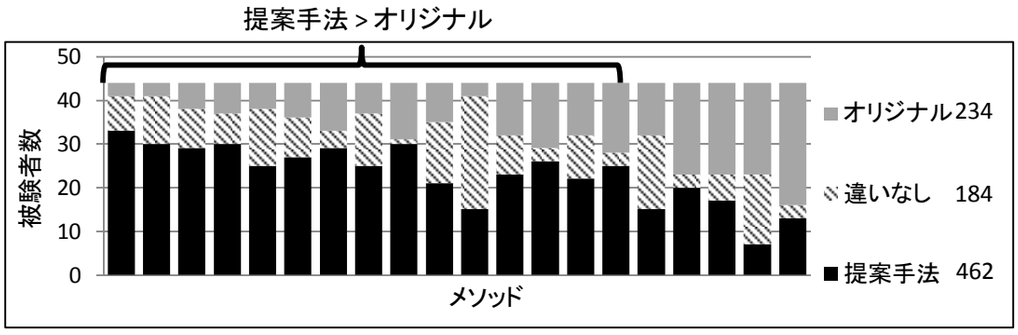


図 3.4　各対象メソッドに対する回答の内訳

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可読性を向上させる文の並べ替えを行うことができたといえる．

3.4.3　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．これら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．
- 同一のオブジェクトに対して，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被験者が最も多かったメソッドである．図中の矢印は DU チェインを表し，そのラベルは DU チェイン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変数を定義していた二つの文が，その変数が参照される直前へ移動するという文の並べ替えが行われた．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二つの変数名が類似している，および，変数 mContent が指すオブジェクトに対して連続してメソッド呼び出しが行われている，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．

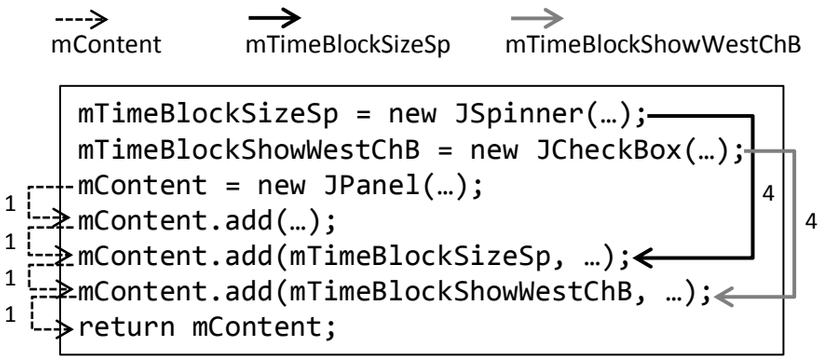
提案手法による文の並べ替えを行うことで，これらの類似した文の構造が保たれていない例がいく

表 3.1　Java を用いたプログラミングの経験

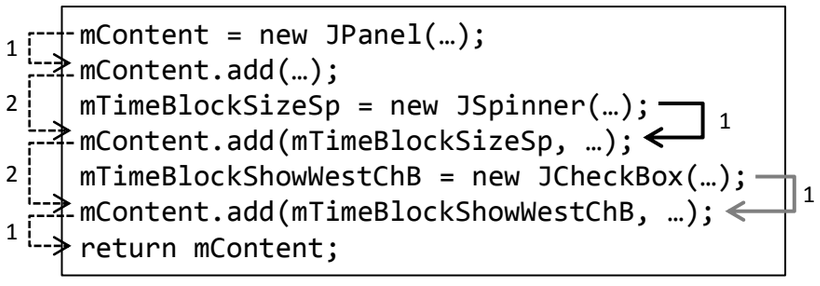
内訳	人数
全く使ったことがない	1 名
1,000 行未満	7 名
1,000〜10,000 行程度	23 名
10,000 行以上	13 名

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



(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

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可読性に貢献するものである可能性がある．

3.5　提案手法の拡張

実験の結果，ソースコードの可読性を向上させるために文の並べ替えを行う場合，変数の定義と参照の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べ替えを行うという手法の拡張を行った．

3.5.1　類似した文の並びの識別

拡張手法では，2 章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応用して類似した文の並びを識別する．繰り返し構造とは定義 2.3.1 の通りである．2 章におけるノードの類似判定基準では，表 2.1 の通り，子を持たないノードは「文の種類と重みが同じ」であること

を基準としている．しかし本拡張手法では，次の 3 種類の文については「文が類似する」という判定基準を追加する．その判定基準は以下の通りである．なお，Java におけるこれらの文の記述形式は表 3.3 の通りである．

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（以降，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，複数の変数が一つの文の中で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，ただ一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，左辺に現れる変数名が類似していれば類似する文とみなす．

変数宣言文と代入文については,「変数名が類似する」ことの判定を行う．この判定には Levenshtein 距離を用いた．Levenshtein 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 Levenshtein 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 Levenshtein 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．

$$Ld(str1, str2) \leq threshold \times max(len(str1), len(str2)) \tag{3.2}$$

ただし，$len(str)$ は文字列 str の長さを表し，$max(len(str1), len(str2))$ は，二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．

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 = . . . ;

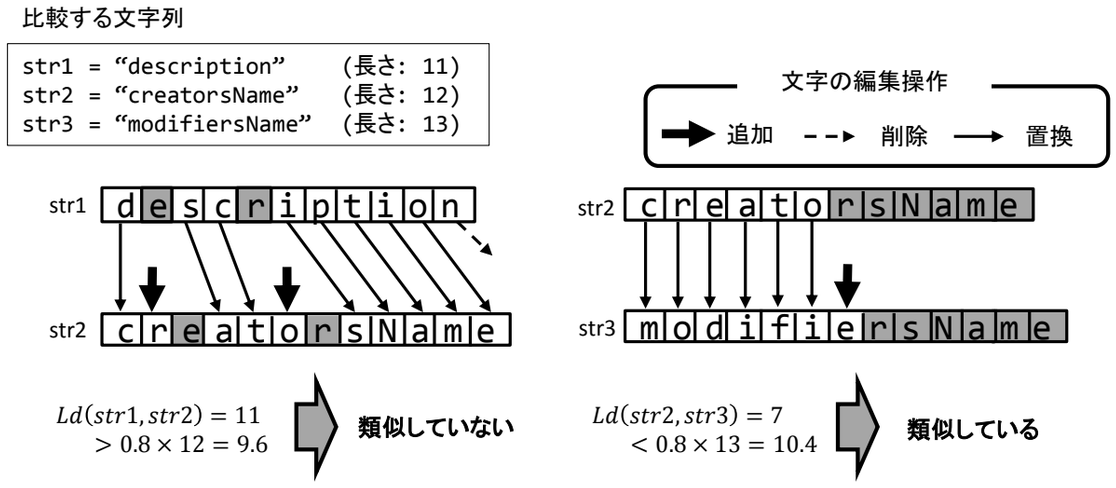


図 3.6　文字列の類似判定の例

Levenshtein 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.

3.5.2　拡張手法の確認

3.4.1 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 のメソッドのうち二つのメソッドで，並べ替えが行われなくなった．それらのメソッドは，3.4.2 節の実験結果において，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.

3.6　提案手法拡張後の評価実験

類似構造を考慮した拡張手法による並べ替えが，考慮しなかった元の手法による並べ替えに比べて，ソースコードの可読性により貢献するかどうか評価するため，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.

3.6.1　準備

実験対象のソフトウェアに対して，類似構造を考慮しない元の提案手法と，類似構造を考慮した拡張提案手法をそれぞれ適用し，二通りの並べ替え結果を得た．このうち，両方の提案手法で文の並べ替えが行われ，かつその結果が異なっていたメソッドの中から 5 つを抽出してアンケート調査を行った．このアンケートでは，各被験者は 5 つのメソッドに対して，元の提案手法によって並べ替えた後のソースコードと，拡張提案手法によって並べ替えた後のソースコードを比較し，どちらが理解しや

すいかを判定した．アンケートの準備は以下の手順で行った．

手順１　5 つのメソッドについて，二通りの並べ替え後のソースコードからコメントおよび空行を取り除く．また，二通りの並べ替え後のソースコード間でインデントや改行位置などのフォーマットを統一する．

手順２　各メソッドにつき，類似構造を考慮した拡張提案手法による並べ替え結果を A，考慮しない元の提案手法による並べ替え結果を 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

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，3.4 節の実験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，表 3.5 の通りである．

また，3.4 節の実験結果における考察に関して知見を得るために，上記のアンケートと合わせて，「文の並びと理解のしやすさ」に関する自由記述のアンケート（任意回答）も実施した．その結果，本実験に参加した 31 名の被験者のうち 8 名の被験者から回答を得られた．

3.6.2　二通りの並べ替え結果に対するアンケート結果

実験結果を図 3.7 に示す．グラフ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 については，多くの被験者が類似構造を考慮した並べ替え結果の方が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分かれる結果となった．また，内訳の合計値を比べると，類似構造を考慮しない並べ替え結果の方が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．

3.6.3　並べ替え結果に対する考察

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．

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

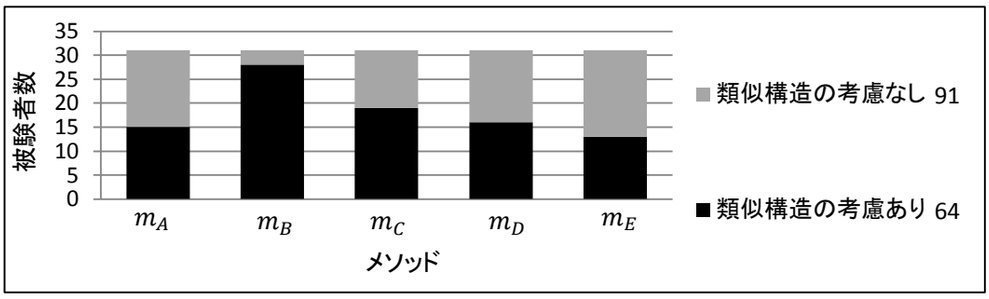
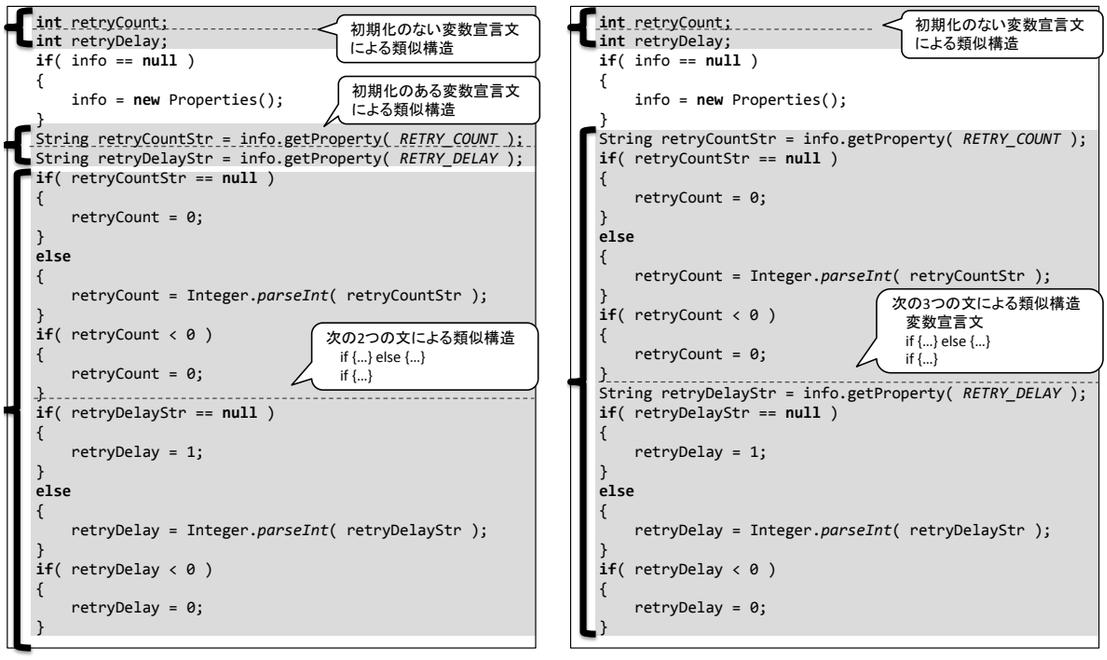


図 3.7　各対象メソッドに対する回答の内訳



(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 の並べ替え結果

メソッド m_A について

メソッド m_A に対する二通りの並べ替え結果を図 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，三箇所の類似構造が特定された．三箇所の類似構造は，いずれも二組の文（または文の並び）によって構成されている．また，三箇所目の類似構造の中には，他二箇所の類似構造の中で宣言された変数が出現している．図 3.8(a) に示すように，類似構造を考慮した並べ替えでは，二箇所

目の類似構造を三箇所目の類似構造に近づけることができた．しかし，一箇所目の類似構造を近づけることはできなかった．この原因は，一箇所目の類似構造に含まれる変数宣言文では変数の初期化が行われていない，すなわち変数が定義されていないため，これらの変数宣言文に対して DU チェインが存在せず，提案手法によって並べ替えが行われなかったためである．

また，類似構造を考慮しない並べ替え結果では，図 3.8(b) に示すように，変数 retryDelayStr の宣言文が，その変数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，オリジナルでは二箇所目と三箇所目の類似構造であったものが，変数宣言文と二つの if 文からなる新たな類似構造となっている．二通りの並べ替え結果の両方に類似構造が存在していたことが，被験者による回答が分かれた理由であると考えられる．被験者のうち何名かは，図 3.8(b) のように，変数の宣言と参照という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．このことから，類似構造を考慮した並べ替えを行う場合は，本拡張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を保持するだけではなく，並べ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

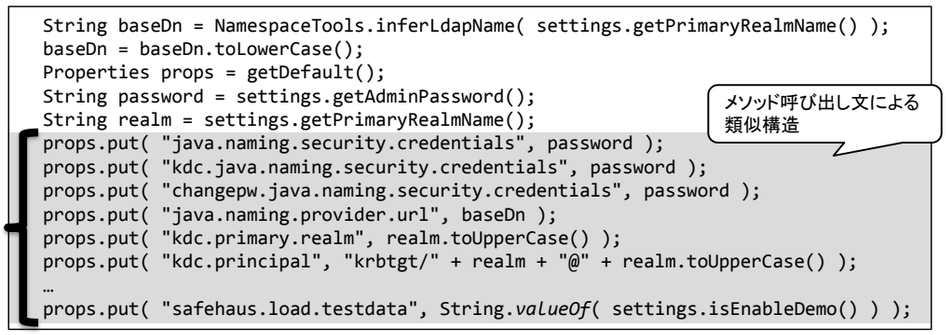
メソッド m_B のオリジナルのソースコードには，メソッド呼び出し文による類似構造が二箇所存在した．類似構造を考慮しない並べ替え結果では，類似構造が一つも残らない結果となった．このメソッドについては，大半の被験者が類似構造を考慮した並べ替え結果の方が理解しやすいと評価している．この結果から，類似構造の存在はソースコードの可読性に貢献している可能性がある．

メソッド m_C について

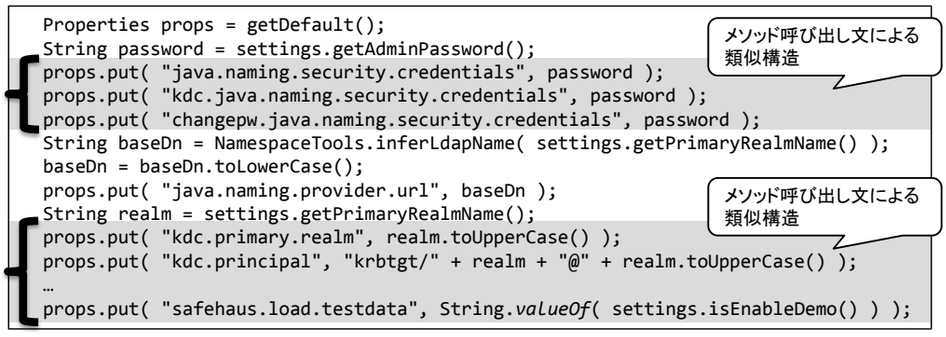
メソッド m_C に対する二通りの並べ替え結果を図 3.9 に示す．メソッド m_C のオリジナルソースコードからは，12 個のメソッド呼び出し文による類似構造が一箇所特定された．このメソッドでは，メソッドの先頭で 4 つの変数が宣言された後，そのうちの一つである変数 props が指すオブジェクトに対するメソッド呼び出しが連続して行われている．類似構造を考慮した並べ替え結果では，先頭の 4 つの変数宣言文の順序が入れ替わったのみであったが，類似構造を考慮しない並べ替え結果では，このうちのいくつかの変数宣言文は，その変数が参照される直前まで移動していた．

類似構造を考慮しない元の提案手法であっても，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と参照の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と参照の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のソースコードでは，連続したメソッド呼び出し中に二箇所の空行が存在する．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，文の並べ替えによって可読性向上の支援を行う際は，そのような意見を反映できる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べ替えが行われており，どちらも被験者による評価結



(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 の並べ替え結果

果は分かれている．

3.6.4　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さについてアンケートには 8 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記述したものであり，一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．また，表 3.6 では回答結果を三つのカテゴリに分けて記載しているが，アンケートではそのようなカテゴリは明記していない．この三つのカテゴリは，被験者が自由記述で回答した内容を元に手作業で分類した結果である．

アンケート結果より，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回答が得られた．また，変数の定義と参照の間の距離については 8 名全員が言及し，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，文の並べ替えを行う際に，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，文の並べ替えは常に全自動で行うべきとは限らないといえる．しかし，変数の定

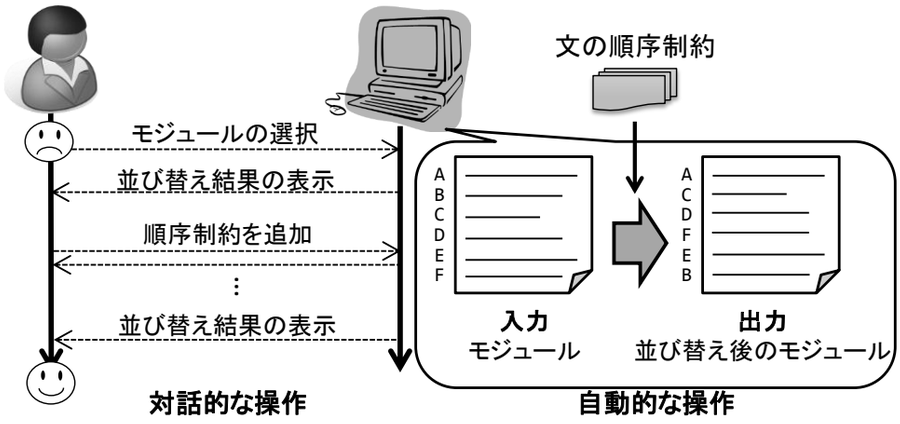


図 3.10　文の並べ替えを利用した対話的なリファクタリング支援環境

義と参照の間の距離を小さくすることや類似構造を保つことで，ソースコードの可読性が向上することも確認できている．従って，文の並べ替えを行う際は，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．

例えば図 3.10 に示すような，対話的なリファクタリング支援環境が有用であると考えられる．ツール上でユーザが可読性を向上させたいメソッドを一つ選択すると，ツールは文の並びに関する初期の制約に従って文を並べ替え，その結果を一つだけ提示する．初期の制約とは，プログラムの振る舞い

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

変数の定義と参照の間の距離について

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

を変えないために最低限守るべき制約である（3.3.4 節を参照）．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べ替え結果を提示することができる．

3.7　妥当性について留意すべき点

3.7.1　被験者について

本実験に参加した被験者は web 上で募集をしており，参加は被験者の意思に委ねられている．そのため，本実験の被験者はソースコードの可読性に関心を持ち，かつ，ある程度のソースコードの読解力も有していると考えられる．そのため，可読性の評価にも偏りが生じている可能性がある．したがって，ソースコードの可読性に対する関心の高さが異なる，より多くの被験者を対象に実験を行った場合は，今回の実験結果とは異なる結果が得られる可能性がある．

3.7.2　実験で用いたプロトタイプについて

3.4 節，及び 3.6 節の実験では，提案手法を簡易実装したプロトタイプを用いることにより自動的にメソッドの並べ替えを行った．しかし，このプロトタイプには提案手法のすべてが実装されているわけではなく，特にオブジェクトの変更に関する情報は取得できていない．たとえば，3.3.1 節に記載している DU チェインのうち，変数が指すオブジェクトの状態が変化したかどうかや，3.3.3 節に記載しているブロック内への文の移動条件のうち，変数が指すオブジェクトの状態変更による再定義が発生したかどうかという点である．同一オブジェクトを複数の変数が参照することができるため，ある変数が指すオブジェクトが，その変数が参照されることなく状態が変更されるケースも発生しうるが，そのようなケースも正確に把握するためには，計算量が膨大となる恐れがあるため本プロトタイプでは未実装であった．そのため，プロトタイプによって並べ替えが行われたメソッドすべてについて，その振る舞いが保たれているわけではない．

3.4 節の実験では，約 3,700 のメソッドのうち，215 のメソッドに対して並べ替えが行われた．実験対象を選定する際は，提案手法により並べ替えられた 215 のメソッドからランダムに一つを抽出し，並べ替えによって振る舞いが変化していないかどうかを手作業により確認した．振る舞いが変わっていないと判断した場合には，そのメソッドを実験対象に加えた．このようにして，ランダムに一つ抽出し，そのソースコードを調査するという作業を，実験対象が 20 になるまで繰り返した．3.6 節の実験では，ランダムに一つ抽出したメソッドについて，二通りの並べ替え結果が異なるかどうか，および並べ替えによって振る舞いが変化していないかどうかを手作業により確認し，実験対象が 5 になるまで繰り返した．

このような手作業による確認作業は，用いたプロトタイプが提案手法の一部を未実装であるため必要であった．もし，提案手法がすべて実装されているツールを用いた場合には，このような手作業による確認作業は必要がない．

3.7.3　メトリクス *totaldistance* について

提案手法では，メトリクス *totaldistance* を計算するために，変数の定義と参照間の文数の単純な和を用いた．しかし，この計算式を距離の対数や二乗の和に変更することで，より可読性が高くなる並べ替えを行える可能性がある．しかしながら，この実験ではそのような並べ替え方法の優劣については評価ができていない．また，並べ替えによる *totaldistance* の変化量が，可読性の評価にどの程度影響を与えたかという分析や，可読性が低いと判断される *totaldistance* のしきい値の評価も本研究では行っていない．

3.7.4　プロトタイプの実行時間について

3.4 節の実験では，作成したプロトタイプを用いて，約 56 分で TVBrowser に含まれているすべてのメソッドに対して並べ替え候補の検出を行った．しかし，このような並べ替えの候補検出は本研究が目指しているあるべき利用状況とは異なる．可読性向上のための並べ替えは，図 3.10 に示したように，ユーザが選択したメソッドに対してのみ実施することが望ましいと考えている．そのような利用状況においては，並べ替え候補提示に必要とする時間は実験で得られた値よりも大幅に短くなる．

また，すでに述べたように実験で用いたプロトタイプは提案手法の一部が未実装である．すべての機能を実装した場合には，現在のプロトタイプに比べてソースコードの解析に要する時間が長くなる．更に，並べ替え対象となるソースコードの特徴によっては，解析時間が増大する恐れがある．例えば提案手法において着目中のブロック内の文を並べ替える際，順序制約を満たす全ての文の並びを生成するため，並べ替え対象となるブロックの中に多くの文が存在し，かつ順序制約が少ない場合は，非常に多くの並びが生成されてしまい，並べ替え結果を得るまでに長い時間を要する可能性がある．

3.8　まとめと今後の課題

本研究では，変数の定義と参照の間の距離に着目して，ソースコード中の文を並べ替える手法を提案した．一つの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ところ，約 3,700 の並べ替え可能なメソッドに対し，215 のメソッドを並べ替えることができた．また，並べ替えが行われたメソッドのうち，20 個のメソッドを対象に，オリジナルのソースコードと提案手法適用後のソースコードの可読性を 44 名の被験者が比較した．その結果，提案手法を適用することで多くのメソッドの可読性が向上することが確認できた．更に，可読性に影響を与える文の並びには，変数の定義と参照の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，本研究で用いた提案手法のプロトタイプに実装されていない機能を実装し，並べ替えを行ってもプログラムの振る舞いを保つよう改善すること，および，図 3.10 に示すような対話的な文の並べ替え支援環境を構築することが挙げられる．

第 4 章

ソースコードに対する欠陥限局の適合性計測

4.1　導入

ソフトウェア開発において，デバッグは多大な労力とコストを必要とする作業である．デバッグ作業がソフトウェア開発コストの過半数を占めるという報告もある [25, 26]．このため，デバッグ支援に関する研究が盛んに行われている．

デバッグ支援の研究分野の一つに欠陥限局技術がある．欠陥限局とは，欠陥が含まれている箇所を推測する技術である．中でも近年，実行経路情報を用いた欠陥限局（Spectrum-Based Fault Localization，以降，SBFL）に関する研究が盛んに行われている [42]．SBFL は，失敗するテストケースによって実行される文は欠陥を含む可能性が高いという考えに基づき，テストケースごとの成否と，プログラム中で実行される文の情報（以下，実行経路情報）を用いて，欠陥を含む文を推測する技術である．SBFL 技術では，プログラム中の各文に対して欠陥を含む可能性（以降，疑惑値）が数値化される．欠陥を含む文の疑惑値が他の文と比べてより高い値であるほど，SBFL 結果がより正確であると言える．

ソフトウェアには様々な品質の観点が存在する．ISO/IEC 25010:2011 [11] で定義されているソフトウェア製品の品質モデルでは，八つの品質特性，及び各特性から派生する副特性が定義されている[*1]．品質特性の一つである Maintainability（保守性）の副特性に Analysability（解析性）がある．解析性とは，ソフトウェアの不具合の原因を診断することや，修正すべき箇所を識別することに対する有効性や効率の程度を示す[*2]．

高水準言語にはさまざまな記述方法が用意されており，開発者は自身の好みや組織あるいはプロ

[*1] ISO/IEC 25010:2011 に準ずる JIS X 25010:2013 [12] では，八つの品質特性を機能適合性・性能効率性・互換性・使用性・信頼性・セキュリティ・保守性・移植性と定義している．

[*2] ISO/IEC 25010:2011 における Analysability の定義は以下の通り．
Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

ジェクトの方針に従って，必要な機能の実装方法を決定する．実装方法が変わるとテストケース毎の実行経路情報が変わる可能性があり，更に文の疑惑値やその順位も変わる可能性がある．従って，プログラム自体が SBFL を用いた欠陥箇所の特定にどの程度適しているかという特性を持っていると考えることができる．

そこで本研究では，あるプログラムに対する SBFL を用いた欠陥限局の結果がどの程度正確かを，プログラムの **SBFL 適合性**として提案する．これは，あるプログラムに対する SBFL 適用技術との親和性の一種であり，品質特性である保守性，及びその副特性である解析性に含まれる一つの観点とみなすことができる．ソフトウェアの品質特性の一つの観点として SBFL 適合性を扱うことにより，下記の活動が可能になる．

- 現在のソースコードに対する SBFL 結果がどの程度信頼できるかを事前に把握する．信頼できると判断された場合は，SBFL 技術を利用することにより欠陥限局を行い，その結果を利用して開発者がデバッグ作業を行えばよい．
- SBFL を利用した欠陥限局を行う前に，SBFL 適合性が低いソースコードに対し，SBFL 適合性が向上するよう一時的なソースコード変換を行う．

本研究では，SBFL 適合性の一つの評価指標として，SBFL スコアの計測方法を提案する．具体的には，全てのテストケースを通過するプログラムに対して，ミューテーションテスト [43] の技術を活用し，様々な箇所に意図的に欠陥を発生させ，SBFL を実行する．SBFL によってどの程度正確に欠陥箇所を特定できたかを計測することにより，元のプログラムの SBFL スコアを計測する．

本研究ではリファクタリングを題材に，プログラム構造の違いによる SBFL スコアの計測結果を比較した．その結果，同じ条件分岐先で実行される文の数が少ないほど SBFL スコアが向上することを確認し，SBFL スコアを向上させるプログラム構造の変換例を発見した．

なお，SBFL スコアの計測は全てのテストケースを通過するプログラムを対象としているため，テストに失敗するプログラムに対する SBFL 結果の信頼度を事前に把握したいという場面では，SBFL スコアを活用することができない．しかし，本研究の結果を基に計測方法や評価指標に関する研究を発展させることで，より実用的な場面で SBFL 適合性を取り扱うことができると考えられる．

本研究による貢献は以下の点である．

- プログラムに対する SBFL 適合性を提案
- SBFL 適合性の一つの評価指標である SBFL スコアについて，ミューテーションテストを用いた計測手法を提案
- リファクタリングを題材として，プログラム構造の違いによって SBFL スコアが異なる事例を確認
- SBFL スコアを向上させるプログラム構造の特徴と変換例を発見

4.2 関連研究

プログラム中の欠陥箇所を推測する手法の 1 つとして，SBFL に関する研究が盛んに行われている [42]．SBFL は，各テストケースの成否と実行経路情報を用いて，文ごとの疑惑値，すなわち欠陥である可能性を示す値を算出する．実行経路情報とは，各テストケースがプログラム中のどの文を実行したかという情報である．直感的には，失敗するテストケースが多く通過する文は，成功するテストケースが多く通過する文より欠陥を含む可能性が高いと考えることができる．これまでに多くの SBFL 手法が提案されている [51–53] が，各手法における疑惑値の計算方法はそれぞれ異なる．Abreu らは SBFL 手法で用いられる計算式の有効性を評価するため，二つの SBFL ツール，および分子生物学の分野において用いられる Ochiai の類似係数 [54] を対象に調査した結果，Ochiai の計算式が最も優れていると結論づけている [55]．

SBFL はテストケースごとの実行経路情報に基づくため，テストケースの内容に結果が左右される．失敗するテストケースをもとに他のテストケースを生成することで，より効率的に SBFL を行う研究が行われている [56, 57]．

また，テストスイートの欠陥検出能力を評価する方法として，ミューテーションテストに関する研究が行われている [43]．ミューテーションテストでは，すべてのテストケースを通過するプログラムに対して意図的に変更を加えた大量のプログラムが生成される．変更が加えられたプログラムをミュータントと呼ぶ．各ミュータントに対してテストを実行した際，テストが失敗すれば，そのテストはミュータントの変更箇所，すなわち欠陥を検出する能力があると判断できる．ミューテーションテストは実際のソフトウェア開発においても適用されており，たとえば品質最重視の企業ソフトウェアに対しても，テストスイートの修正及び品質強化に実用的であることが報告されている [58]．

ミュータントを生成するためのプログラムの変換ルールはミューテーション演算子と表現される．典型的なミューテーション演算子は，プログラム中の単一の変数や式に対して置換，挿入，削除を行うものであり，Java などのオブジェクト指向特有の構文を考慮したミューテーション演算子も研究されている [59, 60]．ミューテーションテストを行うためのツールも多く存在し [61–63]，それぞれのツールは，適用可能なミューテーション演算子の種類とその実装方法，処理性能などに違いがある [64]．

ミューテーションテストを SBFL に応用する研究も行われている．ミューテーションテストでは意図的に変更が加えられるため，その変更箇所，すなわち欠陥の混入箇所は明らかである．欠陥の箇所が未知であるプログラムと，そのプログラムから生成された特定のミュータントが，共通のテストケースで失敗する場合，欠陥の箇所も共通である可能性が高い．この考えに基づき，欠陥限局の精度を向上させる手法が提案されている [65, 66]．

4.3 SBFL 適合性

本研究では，プログラム自体が SBFL にどの程度適しているかという特性を持っていると考え，この特性を **SBFL 適合性**として提案する．プログラムの機能やテストスイートが共通であっても，構

プログラム (入力: a, b)	*susp*	t_1	t_2	t_3	t_4
s_1: **boolean** result = **false**;	0.50	✓	✓	✓	✓
s_2: **if** (0 < a)	0.50	✓	✓	✓	✓
s_3: result = **true**;	0.00	✓	✓		
🐞 s_4: **if** (0 <= b) // correct: 0 < b	0.50	✓	✓	✓	✓
s_5: result = **true**;	0.50	✓	✓	✓	✓
s_6: **return** result;	0.50	✓	✓	✓	✓
テスト結果：		P	P	P	F

(a) リファクタリング前

プログラム (入力: a, b)	*susp*	t_1	t_2	t_3	t_4
s'_2: **if** (0 < a)	0.50	✓	✓	✓	✓
s'_3: **return true**;	0.00	✓	✓		
🐞 s'_4: **if** (0 <= b) // correct: 0 < b	0.71			✓	✓
s'_5: **return true**;	0.71			✓	✓
s'_6: **return false**;	-				
テスト結果：		P	P	P	F

(b) リファクタリング後

テストケース	入力 (a, b)	期待値	実際の値
t_1:	(1, 1)	true	true
t_2:	(1, 0)	true	true
t_3:	(0, 1)	true	true
t_4:	(0, 0)	false	true

(c) テストスイート

図 4.1　構造の異なる二つのプログラムの SBFL 結果

造が異なることで SBFL を用いた欠陥限局の精度に違いが生じることがある．本節では，プログラム構造の違いによる SBFL 適合性の違いについて，具体的事例を用いて説明する．

図 4.1 は，二つの入力値を受け取り，少なくとも一方が正の数であれば true を，そうでなければ false を返す処理を，二通りの記述で表している．図 4.1(a) では結果を変数 result に代入して，最後にまとめて結果を返しているが，図 4.1(b) では結果が確定したタイミングで逐次返している．図 4.1(b) は図 4.1(a) に対して，リファクタリングが適用された状態である．このように return 文によって早く処理を抜ける書き方は early return と呼ばれる [67].

このプログラムのテストスイートには図 4.1(c) を用意した．図 4.1 の各プログラムは同じ条件式に欠陥を含んでいる．図 4.1(a) の s_4 と図 4.1(b) の s'_4 は変数 b が 0 である場合も true を返してしまうため，テストケース t_4 は失敗する．

図 4.1(a)，4.1(b) の右側には，各テストケースの結果（P：通過（成功），F：失敗）と実行経路情報，および文の疑惑値（suspiciousness，図では susp と記載）も記載している[*3]．なお，ここでの文の疑惑値は，既存研究により最も優れていると評価されている Ochiai の計算式を用いて算出している．計算式は後述するが，その文を実行したテストケースのうち最終的に失敗したものが多いほど，

[*3] 用意したテストスイートでは図 4.1(b) の文 s'_6 は実行されないため，文 s'_6 の疑惑値の情報は存在しない．

その文の疑惑値は高くなる．疑惑値のとりうる値は 0 以上 1 以下であり，1 が最も疑惑が強いことを示す．図 4.1 より，欠陥を含む文 s_4，s'_4 は異なる疑惑値であると確認できる．

欠陥箇所が未知である状態において，SBFL 結果をもとに欠陥の箇所を特定しようとする作業を考える．文ごとの疑惑値が算出されているため，作業者は疑惑値の高い順に文を確認する．図 4.1(a) の場合，欠陥を含む文の疑惑値は最も高いものの，同じ疑惑値を持つ文が計 5 つ存在するため，最大で 5 つの文を確認しなければ発見できない．一方で，図 4.1(b) であれば，二つの文を確認すれば欠陥箇所を発見することができる．したがって，図 4.1(a) に比べて図 4.1(b) の方が SBFL 結果の精度が高いといえる．

4.4 SBFL スコア

本研究では，プログラムの SBFL 適合性を評価するために，**SBFL スコア**という指標を提案する．本研究において，SBFL 適合性と SBFL スコアは下記のとおり区別する．

SBFL 適合性： ソフトウェアの品質特性として提案．性質を表すものであり，直接的な数値化はできない．

SBFL スコア： SBFL 適合性の一つの評価指標として提案．具体的な数値を持つ．

SBFL スコアを計測するための基本的なアイデアは，与えられたプログラムに対して意図的に変更を加えたプログラムを複数生成し，その変更箇所を欠陥とみなして，SBFL でどの程度正確に特定できるか計測することである．プログラムに意図的に変更を加える方法として，ミューテーションテストを活用する．本来ミューテーションテストとは，テストスイートがミュータントをどの程度正確に欠陥として検出できるかを計測するが，本手法は，ミュータントに含まれる欠陥箇所を，SBFL がどの程度正確に識別できるかを計測する．

4.4.1 SBFL スコアに影響する要素

プログラム p の SBFL スコアは次の二つの要素に影響を受ける．

- テストスイート T
- ミュータント生成器 G

テストスイートに含まれるテストケースの内容によって SBFL の精度に違いが生じる可能性がある．例として，図 4.1(b) においてテストケースが t_3，t_4 のみである場合を考える．この二つのテストケースの実行経路情報は同一であるため，実行経路情報から計算される疑惑値は，少なくとも実行される文 s'_2，s'_4，s'_5 において同じ値となる．元のテストスイートによる SBFL 結果であれば，文 s'_2 より文 s'_4，s'_5 の方に欠陥が含まれている可能性が高いと判断できる．

本研究では，ミュータントを生成する仕組みをミュータント生成器と呼ぶ．ミューテーションテストにおいて，ミュータントを生成するためのプログラムの変換ルールをミューテーション演算子と表

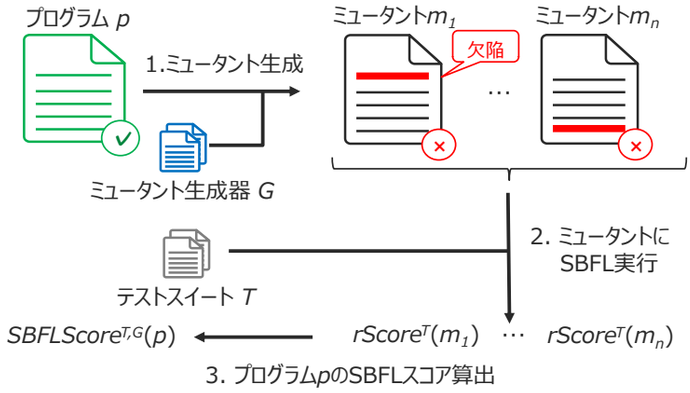


図 4.2　SBFL スコア算出の流れ

現する．ミューテーション演算子の種類は，ミュータント生成器を構成する要素の一つである．なお，複数のミューテーション演算子をプログラムの複数箇所に適用してミュータントを生成することも可能ではあるが，本提案手法におけるミュータントの生成は，一つのミューテーション演算子をプログラムの一箇所にのみ適用することを前提とする．

4.4.2　SBFL スコアの算出方法

プログラム p の SBFL スコアは，テストスイート T，ミュータント生成器 G に依存するとして，$SBFLScore^{T,G}(p)$ と定義する．SBFL スコアは 0 以上 1 以下の値をとり，値が高いほど SBFL 適合性が高いものとする．

SBFL スコアの算出の流れを図 4.2 に示す．SBFL スコアは次の三つのステップによって算出される．

1. プログラム p から複数のミュータントを生成.
2. 各ミュータントに SBFL を実行し，意図的に発生させた欠陥の疑惑値の順位を算出.
3. 各ミュータントに含まれる欠陥の疑惑値の順位から，SBFL スコアを算出.

ステップ 1. ミュータント生成

プログラム p にミュータント生成器 G を用いて，ミュータントを生成する．このとき，各ミュータントは元のプログラムに対して一箇所だけが変更されるよう生成する．この変更が加えられた箇所を欠陥として扱うが，この時点ではテストを実行していないため，変更箇所が本当に欠陥となるかどうかは分からない．生成されたミュータントの集合を $M^G(p)$ と定義する．

ステップ 2. ミュータントに SBFL を実行

生成された各ミュータントに対して，テストスイート T を用いて SBFL を実行する．T に含まれるすべてのテストケースを通過したミュータントが存在した場合は，そのミュータントの欠陥を検出

できるようテストスイートを修正のうえ，改めて SBFL を実行し，すべてのテストケースを通過するミュータントが存在しない状態とする[*4]．

$M^G(p)$ に含まれる各ミュータントに対し，文ごとの疑惑値を算出し，順位付けを行う．ここで，あるミュータント $m \in M^G(p)$ に含まれる各文 s について，以下を定義する．

- $susp^T(s)$：文 s の疑惑値
- $rank^T(s)$：文 s の疑惑値の順位
- $rScore^T(s)$：文 s の疑惑値の正規化順位

疑惑値の算出には Ochiai の計算式を用いる．テストスイート T を実行したときのある文 s の疑惑値 $susp^T(s)$ は，Ochiai の計算式によって以下のとおり算出される．以下の式において，$fail^T(s)$ は文 s を実行した失敗テストケースの数，$pass^T(s)$ は文 s を実行した成功テストケースの数，$totalFail^T$ は失敗テストケースの総数である．

$$susp^T(s) = \frac{fail^T(s)}{\sqrt{totalFail^T \times (fail^T(s) + pass^T(s))}} \tag{4.1}$$

次に疑惑値の順位 $rank^T(s)$ を算出する．ある文の疑惑値の順位は，疑惑値の高い順に文を並べた際最大で確認しなければならない文の総数とする．たとえば，疑惑値 1.0 の文が二つ，0.9 の文が一つ存在する場合は，疑惑値 1.0 の文はどちらも 2 位と扱い，疑惑値 0.9 の文は 3 位とする．

疑惑値の順位は，順位の母集団となる文の総数により異なる価値を持つ．たとえば，10 個の文のうちの 10 位と，100 個の文のうちの 10 位とでは，後者の方が順位としての価値が高い．そこで，各文が文全体の中でどの程度上位に登場するかを示すため，順位を 0 以上 1 以下の範囲に線形に正規化する．文 s の正規化順位 $rScore^T(s)$ は以下のとおり算出する．1 が最も順位が高く，0 が最も順位が低いことを示している．以下の式において，$totalStatements^T$ はテストスイート T によって実行される文の数である．

$$rScore^T(s) = 1 - \frac{rank^T(s) - 1}{totalStatements^T - 1} \tag{4.2}$$

また，ミュータント m に含まれる欠陥の疑惑値の正規化順位を $rScore^T(m)$ と定義する．各ミュータントに含まれる欠陥はただ一つの文であるため，この値はミュータントごとに一意である．ミュータント m の欠陥を含む文を ${s^m}_{fault}$ とすると，$rScore^T(m)$ は文 ${s^m}_{fault}$ の疑惑値の正規化順位である．

$$rScore^T(m) = rScore^T({s^m}_{fault})$$

ミュータントの $rScore$ が高いほど，そのミュータントの SBFL 結果の精度が高い，すなわち欠陥

[*4] プログラムの記述次第では，どのようなテストケースも通過してしまう可能性がある．たとえば `a = b + c; a = d;` という二つのプログラム文が並んで記述されている場合，一文目の代入結果が二文目によって上書きされるため，一文目の代入式に変更を加えてもテストケースで検出することができない．このような場合はプログラムを見直してステップ 1 から実行し直す必要がある．

プログラム (入力: a, b)	*susp*	*rank*	*rScore*
s_1: **boolean** result = **false**;	0.50	5	0.20
s_2: **if** (0 < a)	0.50	5	0.20
s_3: result = **true**;	0.00	6	0.00
s_4: **if** (0 <= b) // correct: 0 < b	0.50	5	0.20
s_5: result = **true**;	0.50	5	0.20
s_6: **return** result;	0.50	5	0.20

(a) リファクタリング前

プログラム (入力: a, b)	*susp*	*rank*	*rScore*
s'_2: **if** (0 < a)	0.50	3	0.33
s'_3: **return true**;	0.00	4	0.00
s'_4: **if** (0 <= b) // correct: 0 < b	0.71	2	0.67
s'_5: **return true**;	0.71	2	0.67
s'_6: **return false**;	-	-	-

(b) リファクタリング後

図 4.3　ミュータントへの SBFL 実行結果

箇所を正確に特定できることを意味する．例として，図 4.1 のプログラムをミュータントと見立てた場合の SBFL 結果を図 4.3 に示す．欠陥を含む文 s_4，s'_4 の *rScore*，およびこのミュータント自体の *rScore* はそれぞれ 0.20 と 0.67 である．4.3 節で述べたとおり，リファクタリング後の方が SBFL 結果の精度が高いという事実を，*rScore* によって示すことができている．

ステップ 3.SBFL スコアの算出

プログラム p から生成された各ミュータントの *rScore* の平均値を SBFL スコアとし，以下の式で算出する．

$$SBFLScore^{T,G}(p) = \frac{1}{|M^G(p)|} \sum_{m \in M^G(p)} rScore^T(m)$$

4.5　実験

Java を対象として提案手法を実装し，プログラム構造の違いにより SBFL スコアがどのように変化するか確認するための実験を行った．

4.5.1　準備

本実験で用いるミューテーション演算子は，オープンソースのミューテーションテストツールである PIT [68] を参考とした．PIT の Web サイトではミューテーション演算子のカテゴリと内容が公開されている．PIT 自体はバイトコードとしてのミュータントしか生成しないため，ソースコードとし

てミュータントを生成するよう，PIT のデフォルトカテゴリに含まれる 11 種類のミューテーション演算子を本実験のために実装した．本実験で用いるミューテーション演算子を表 4.1 に示す．本研究では表 4.1 の括弧内の表記を各ミューテーション演算子の略称として使用する．

本研究では，プログラム構造の違いとしてリファクタリングを題材とした．リファクタリングには様々なパターンが存在する [18] が，今回は，「条件式の簡素化」に分類される 8 種類のリファクタリングパターンの中から選定した．このカテゴリから選定した理由は，条件式の分岐によってテストケースごとの実行経路情報が変化するため，リファクタリングによって SBFL 結果が変化しやすいと考えたためである．また，本実験では SBFL を Java メソッド単位で実行し，対象となるメソッド内で文の順位を計算するよう提案手法を実装している．そのため，複数のメソッドにまたがるリファクタリングパターンを題材とすることができない．そのようなリファクタリングパターンについては，挙動の似ている別のリファクタリングパターンで代用した．たとえば，「条件記述の分解」は本来，条件文中の記述を他のメソッドに切り出すリファクタリングであるが，メソッドではなく変数に切り出す「変数の切り出し」で代用した．また，「条件式の統合」は本来，一連の条件式を単一の条件式に統合し，他のメソッドに切り出すリファクタリングであるが，ここでは統合するのみとした．代用が難しい 3 種類のリファクタリングパターンは対象から除外し，残った 5 種類のリファクタリングパターンを実験対象の題材として選定した．選定したリファクタリングパターンを表 4.2 に示す．

各プログラムはミューテーション演算子がなるべく多くの箇所に適用されるよう配慮して作成した．詳細は後述するが，本実験ではリファクタリング前後で同じ記述箇所に発生させた欠陥に対して特定のしやすさの変化を確認する．欠陥箇所に偏りがあると，実験の評価に影響を与える可能性があるため，このような配慮を行った．たとえば，条件式に boolean 型の変数だけを書くとミューテーション演算子が適用されないため，`var == true` のように関係演算子を用いて NC 演算子が適用されるようにしている．

テストスイートは，C2 カバレッジ（条件網羅）が 100% となるように生成した．これは，なるべく

表 4.1　実験で用いるミューテーション演算子

ミューテーション演算子	説明	変換例	
		変換前	変換後
(CB) Conditional Boundary	関係演算子の境界を変更	`a<b`	`a<=b`
(INC) Increments	インクリメントとデクリメントを入替	`n++`	`n--`
(INV) Invert Negatives	負の数を正の数に置換	`-n`	`n`
(MA) Math	算術演算子を置換	`a+b`	`a-b`
(NC) Negate Conditionals	関係演算子を置換	`a==b`	`a!=b`
(VM) Void Method Calls	戻り値のないメソッド呼び出しを削除	`method();`	`;`
(PR) Primitive Returns	プリミティブ型の戻り値を 0 に置換	`return 5;`	`return 0;`
(ER) Empty Returns	戻り値の型に応じて空を表す値に置換	`return "str";`	`return "";`
(FR) False Returns	戻り値を false に置換	`return true;`	`return false;`
(TR) True Returns	戻り値を true に置換	`return false;`	`return true;`
(NR) Null Returns	戻り値を null に置換	`return object;`	`return null;`

多くのテストケースを用いることで，各プログラム文の疑惑値を分散させるためである．ただし，提案手法により生成されるミュータントによっては，元のプログラムのテストスイートの C2 カバレッジが 100% にならない可能性がある．たとえば，プログラムの条件式に算術演算子が含まれる場合，MA 演算子の適用により条件式中の境界値が変わる可能性がある．そのため，まずは提案手法のステップ 1 においてミュータントの生成を行い，すべてのミュータントの C2 カバレッジが 100% となるテストスイートを手作業で作成した．

対象プログラムを図 4.4〜4.8 に示す．このうち，図 4.4〜4.6 は以降の節にて詳細を述べるため，生成されたミュータントに関する情報も掲載している．図 4.7，4.8 は参考として 4.5 節の末尾に掲載している．

4.5.2 結果と考察

SBFL スコアの計測結果を表 4.2 の右側に示す．ケース 1〜3 はリファクタリング前の SBFL スコアが高く，ケース 4，5 ではリファクタリング後の SBFL スコアが高いという結果となった．

例としてケース 5，1，3 のプログラムと生成されたミュータントに関する情報を図 4.4，4.5，4.6 に示す．(a) がリファクタリング前，(b) がリファクタリング後であり，それぞれの左側にプログラム，右側に各ミュータントにおける各文の *rScore* を表示している．*rScore* は数値と棒グラフで表示されており，各ミュータントのうち元のプログラムから変更された文，すなわち欠陥を含む文の *rScore* は太字で表現されている．この値は 4.4.2 節で述べたとおりミュータントの *rScore* でもある．たとえば，図 4.4(a) のミュータント m_1 では，文 s_2 に対して NC 演算子が適用されており，*rScore* が 0.67 であることを示している．太字で表現しているミュータントの *rScore* の平均値が *SBFLScore* である．

図 4.4 の文 s_2 と s'_2 のようにリファクタリング前後で互いに対応する文は同じ数字を添字として付与している．たとえば図 4.4 の文 s_2 と s'_2 はリファクタリング前後で変化はなく，図 4.5 の文 s'_{1a} と s'_{1b} はリファクタリングによって文 s_1 から分割されたことを示している．また，図 4.6 の文 $s'_{\{5,7\}}$ は，文 s_5，s_7 がリファクタリングによって一つの文に集約されたことを示している．

また，互いに対応する文，すなわち同じ数字を添字に持つ文に同じミューテーション演算子が適用

表 4.2　対象リファクタリングと SBFL スコアの計測結果

ケース：リファクタリングパターン	*SBFL* スコア	
	前	後
1：変数の切り出し（「条件記述の分解」の代用）	0.81	0.53
2：条件式の統合（メソッド抽出なし）	0.95	0.72
3：重複した条件記述断片の統合	0.69	0.53
4：制御フラグの削除	0.61	0.67
5：ガード節による入れ子条件記述の書き換え	0.83	0.95

				rScore				
ミュータント:	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
ミューテーション演算子:	NC	CB	INV	NC	CB	INV	INV	PR
s_1: **int** result = 0;	0.67	0.50	0.50	0.33	0.33	0.33	0.33	0.67
s_2: **if** (x > 0)	**0.67**	**0.50**	0.50	0.33	0.33	0.33	0.33	0.67
s_3: result = -10;	0.50	1.00	**1.00**	0.00	0.00	0.00	0.00	0.17
s_4: **else if** (y > 0)	0.33	0.00	0.00	**1.00**	**0.83**	0.83	0.83	0.50
s_5: result = -20;	0.00	0.00	0.00	0.83	1.00	**1.00**	0.00	0.00
else	-	-	-	-	-	-	-	-
s_6: result = -30;	0.17	0.00	0.00	0.17	0.00	0.00	**1.00**	0.33
s_7: **return** result;	0.67	0.50	0.50	0.33	0.33	0.33	0.33	**0.67**

(a) リファクタリング前 ($SBFLScore = 0.83$)

				rScore						
ミュータント:	m'_1	m'_2	m'_3	m'_4	m'_5	m'_6	m'_7	m'_{8a}	m'_{8b}	m'_{8c}
ミューテーション演算子:	NC	CB	INV	NC	CB	INV	INV	PR	PR	PR
s'_2 : **if** (x > 0)	**1.00**	**0.75**	0.75	0.50	0.50	0.50	0.50	0.50	0.75	0.50
$s'_{\{3,7\}}$: **return** -10;	0.75	1.00	**1.00**	0.00	0.00	0.00	0.00	0.00	**1.00**	0.00
s'_4 : **if** (y > 0)	0.50	0.00	0.00	**1.00**	**0.75**	0.75	0.75	0.75	0.00	0.75
$s'_{\{5,7\}}$: **return** -20;	0.00	0.00	0.00	0.75	1.00	**1.00**	0.00	0.00	0.00	**1.00**
$s'_{\{6,7\}}$: **return** -30;	0.25	0.00	0.00	0.25	0.00	0.00	**1.00**	**1.00**	0.00	0.00

(b) リファクタリング後 ($SBFLScore = 0.95$)

図 4.4　ケース 5：ガード節による入れ子条件記述の書き換え

されて生成されたミュータントを，ミュータントのペアと呼ぶ．たとえば，図 4.4 の m_1 と m'_1 は，文 s_2 と s'_2 に NC 演算子が適用されたミュータントのペアである．ペアとなるミュータントは同じ数字を添字として付与している．以降，ミュータントのペアは，$\langle m_1, m'_1 \rangle$ と表現する．ペアとなるミュータントの *rScore* を比較することで，同じ記述箇所に発生した欠陥の正規化順位，すなわちその欠陥箇所の特定のしやすさが，リファクタリング前後でどれだけ変化するかを確認することができる．

SBFL スコアが向上した例

例としてケース 5 について考察する．ケース 5 で適用した「ガード節による入れ子条件記述の書き換え」とは，後続処理の対象外となる条件が満たされれば return する処理を先頭に記述することで，早く処理を終了させるよう書き換えるリファクタリングである．4.3 節で用いたリファクタリング例と同様，このような書き方は early return とも呼ばれる．これらはソースコードのネストが深くなることを抑制する効果がある．

図 4.4 よりミュータントのペアを比較すると，$\langle m_1, m'_1 \rangle$，$\langle m_2, m'_2 \rangle$，$\langle m_8, m'_{8a} \rangle$，$\langle m_8, m'_{8b} \rangle$，$\langle m_8, m'_{8c} \rangle$ はリファクタリング後に *rScore* が向上，$\langle m_5, m'_5 \rangle$ はリファクタリング後に *rScore* が低下，それ以外は変わらなかった．リファクタリング前の各ミュータントにおいて，文 s_1，s_2，s_7

は同じ *rScore* であることに着目する．この文が欠陥箇所であるミュータントは m_1，m_2，m_8 であり，リファクタリング後に *rScore* が向上したミュータントのペアに含まれる．*rScore* の計算過程を確認したところ，これらの文の疑惑値もまた互いに同じ値であった．4.4.2 節で述べたとおり，提案手法では同じ疑惑値を持つ文が多いほど，その文の順位が下がるような順位付けを行っている．文 s_1，s_2，s_7 はどのテストケースにおいても必ず実行される文であるため同じ疑惑値であったが，リファクタリング後は return 文が挿入されたことで，必ず実行される文が s'_2 のみとなったため，順位および *rScore* が向上したと考えられる．

式 (4.1) において，実行されるテストケースが共通である文，すなわち同じ条件分岐先[*5]で実行される文は，つねに同じ疑惑値を持つ．同じ条件分岐先で実行される文を明確にするため，条件分岐先ごとに文を分類した．その結果を表 4.3 に示す．この表において，C(s_2, s_4) は文 s_2 と s_4 にそれぞれ含まれる条件式の結果に応じた条件分岐先を意味する．条件式の結果のとりうる値は，{T, F, *} の三通りであり，それぞれ真，偽，影響なしを示す．たとえば C(*, *) は，いずれの条件式にも影響を受けない条件分岐先であり，リファクタリング前は文 s_1，s_2，s_7 が，リファクタリング後は s'_2 が該当する．表 4.3 より，C(*, *) に該当する文の数はリファクタリングによって 3 から 1 に減っており，それ以外は数に変わりはない．同じ条件分岐先の文が少ないことは，同じ疑惑値を持つ文が少ないことを意味しているため，同じ条件分岐先の文が少ないほど，SBFL スコアが向上すると考えることができる．

SBFL スコアが低下した例

例としてケース 1，3 について考察する．ケース 1 では，条件文中の記述を変数に切り出す「変数切り出し」リファクタリングを題材とした．条件分岐先ごとに文を分類したところ，表 4.4 のとおり，C(*, *) に該当する文の数が増加していることが確認できた．図 4.5 より，C(*, *) に該当する文が欠陥箇所であるミュータントのペアにおいては，リファクタリング後のミュータントの方が，*rScore* が低いことが読み取れる．条件式を別の文に切り出したことで同一条件分岐先で実行される文の数が増

表 4.3　条件分岐先ごとの文の分類（ケース 5）

(a) リファクタリング前

条件分岐先 C(s_2, s_4)	該当文
C(*, *)	$\{s_1, s_2, s_7\}$
C(T, *)	$\{s_3\}$
C(F, *)	$\{s_4\}$
C(F, T)	$\{s_5\}$
C(F, F)	$\{s_6\}$

(b) リファクタリング後

条件分岐先 C(s'_2, s'_4)	該当文
C(*, *)	$\{s'_2\}$
C(T, *)	$\{s'_{\{3,7\}}\}$
C(F, *)	$\{s'_4\}$
C(F, T)	$\{s'_{\{5,7\}}\}$
C(F, F)	$\{s'_{\{6,7\}}\}$

[*5] ここでは，文 s_1，s_2，s_7 のように必ず実行される文も，条件分岐に影響を受けない一つの条件分岐先とみなしている．

	rScore						
ミュータント:	m_1	m_2	m_3	m_4	m_5	m_6	m_7
ミューテーション演算子:	NC	CB	INC	NC	CB	INC	PR
s_1: **if** (0 < n)	**0.67**	**0.50**	0.50	0.50	0.25	0.25	0.75
s_2: n--;	0.33	1.00	**1.00**	0.00	0.00	0.00	0.25
s_3: **else if** (n < 0)	0.00	0.00	0.00	**1.00**	**0.75**	0.75	0.00
s_4: n++;	-	0.00	0.00	0.25	0.75	**1.00**	0.25
s_5: **return** n;	0.67	0.50	0.50	0.50	0.25	0.25	**0.75**

(a) リファクタリング前 ($SBFLScore = 0.81$)

	rScore						
ミュータント:	m'_1	m'_2	m'_3	m'_4	m'_5	m'_6	m'_7
ミューテーション演算子:	NC	CB	INC	NC	CB	INC	PR
s'_{1a} : **boolean** f1 = (0 < n);	**0.40**	**0.33**	0.33	0.33	0.17	0.17	0.50
s'_{3a} : **boolean** f2 = (n < 0);	0.40	0.33	0.33	**0.33**	**0.17**	0.17	0.50
s'_{1b} : **if** (f1)	0.40	0.33	0.33	0.33	0.17	0.17	0.50
s'_2 : n--;	0.20	1.00	**1.00**	0.00	0.00	0.00	0.17
s'_{3b} : **else if** (f2)	0.00	0.00	0.00	1.00	0.83	0.83	0.00
s'_4 : n++;	-	0.00	0.00	0.17	0.83	**1.00**	0.17
s'_5 : **return** n;	0.40	0.33	0.33	0.33	0.17	0.17	**0.50**

(b) リファクタリング後 ($SBFLScore = 0.53$)

図 4.5　ケース 1：変数の切り出し

え，SBFL スコアが低下したと考えられる．

ケース 3 で適用した「重複した条件記述断片の統合」は，if-else の両方で実行される文を if 文の外に切り出すリファクタリングである．条件分岐先ごとに文を分類した結果を表 4.5 に示す．C(*) に該当する文の数は増加し，C(T)，C(F) に該当する文の数は減少しており，ケース 5 と 1 の両方の事象が起こっている．図 4.6 よりミュータントのペアを比較すると，リファクタリングによって *rScore*

表 4.4　条件分岐先ごとの文の分類（ケース 1）

(a) リファクタリング前

条件分岐先 C(s_1, s_3)	該当文
C(*, *)	$\{s_1, s_5\}$
C(T, *)	$\{s_2\}$
C(F, *)	$\{s_3\}$
C(F, F)	$\{s_4\}$

(b) リファクタリング後

条件分岐先 C(s_{1b}, s_{3b})	該当文
C(*, *)	$\{s'_{1a}, s'_{3a}, s'_{1b}, s'_5\}$
C(T, *)	$\{s'_2\}$
C(F, *)	$\{s'_{3b}\}$
C(F, F)	$\{s'_4\}$

	rScore						
ミュータント:	m_1	m_2	m_3	m_4	m_5	m_6	m_7
ミューテーション演算子:	NC	CB	MA	MA	MA	MA	PR
s_1: **int** result = 0;	0.57	0.29	0.29	0.29	0.29	0.29	0.57
s_2: **int** tmp = 0;	0.57	0.29	0.29	0.29	0.29	0.29	0.57
s_3: **if** (x > 0) {	**0.57**	**0.29**	0.29	0.29	0.29	0.29	0.57
s_4: tmp = y * 2;	0.29	0.86	**0.86**	0.00	0.86	0.00	0.00
s_5: result = y + tmp;	0.29	0.86	0.86	0.00	**0.86**	0.00	0.00
} **else** {							
s_6: tmp = y * 3;	0.00	0.00	0.00	**0.86**	0.00	0.86	0.29
s_7: result = y + tmp;	0.00	0.00	0.00	0.86	0.00	**0.86**	0.29
}							
s_8: **return** result;	0.57	0.29	0.29	0.29	0.29	0.29	**0.57**

(a) リファクタリング前 ($SBFLScore = 0.69$)

	rScore					
ミュータント:	m'_1	m'_2	m'_3	m'_4	$m'_{\{5,6\}}$	m'_7
ミューテーション演算子:	NC	CB	MA	MA	MA	PR
s'_1 : **int** result = 0;	0.33	0.17	0.17	0.17	0.33	0.33
s'_2 : **int** tmp = 0;	0.33	0.17	0.17	0.17	0.33	0.33
s'_3 : **if** (x > 0)	**0.33**	**0.17**	0.17	0.17	0.33	0.33
s'_4 : tmp = y * 2;	0.17	1.00	**1.00**	0.00	0.00	0.00
else						
s'_6 : tmp = y * 3;	0.00	0.00	0.00	**1.00**	0.17	0.17
$s'_{\{5,7\}}$: result = y + tmp;	0.33	0.17	0.17	0.17	**0.33**	0.33
s'_8 : **return** result;	0.33	0.17	0.17	0.17	0.33	**0.33**

(b) リファクタリング後 ($SBFLScore = 0.53$)

図 4.6　ケース 3：重複した条件記述断片の統合

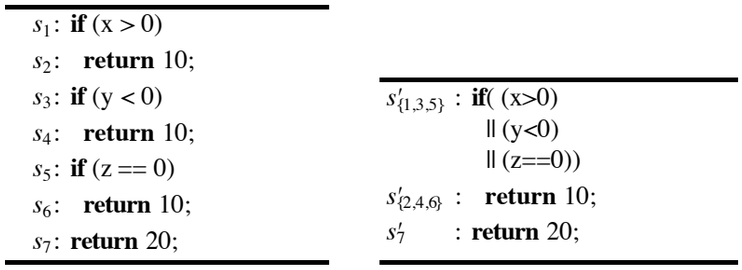
表 4.5　条件分岐先ごとの文の分類（ケース 3）

(a) リファクタリング前

条件分岐先 C(s_3)	該当文
C(*)	$\{s_1, s_2, s_3, s_8\}$
C(T)	$\{s_4, s_5\}$
C(F)	$\{s_6, s_7\}$

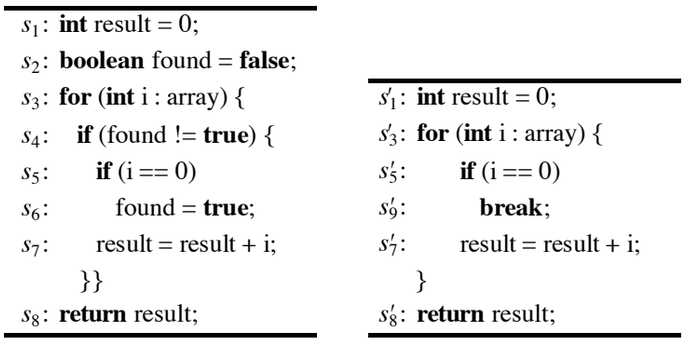
(b) リファクタリング後

条件分岐先 C(s'_3)	該当文
C(*)	$\{s'_1, s'_2, s'_3, s'_{\{5,7\}}, s'_8\}$
C(T)	$\{s'_4\}$
C(F)	$\{s'_6\}$



(a) リファクタリング前　　(b) リファクタリング後

図 4.7　ケース 2：条件式の統合



(a) リファクタリング前　　(b) リファクタリング後

図 4.8　ケース 4：制御フラグの削除

が低下したミュータントのペアの方が，増加したミュータントのペアよりも多い．これは，リファクタリング後に文の数が増加した C(*) に多くの文が該当していることが理由であると考えられる．また，$\langle m_5, m'_{\{5,6\}} \rangle$ と $\langle m_6, m'_{\{5,6\}} \rangle$ のペアは，他のペアと比べてリファクタリング前後の $rScore$ の差が大きい．これは，このミュータントの欠陥を含む文が該当する条件分岐先が，リファクタリングによって C(T)， C(F) から C(*) に変更となり，それらの分岐先で実行される文の数が 2 から 5 に大きく増えたことが理由であると考えられる．このことから，条件分岐先の文の数が少ない方から多い方へ文が移動したことが，SBFL スコアが低下した原因と考えることができる．逆に，図 4.6(b) から 4.6(a) への変換のように，多い方から少ない方へ文の移動を行うことで，SBFL スコアを向上させることができるといえる．

4.5.3　まとめ

実験結果より，同一の条件分岐先で実行される文の数が少ないほど，SBFL スコアが向上することを確認した．また，これを実現するためのプログラム構造の変換方法として，以下の手段が有効であると考えられる．

- early return を用いて条件分岐を早く終了させる．

- 同一の条件分岐先の文の数が多い方から少ない方へ文を移動する．

4.6 妥当性について留意すべき点

本提案手法では，生成されたミュータントの *rScore* の平均値を SBFL スコアとした．しかし，ミュータントの *rScore* に外れ値が含まれていたり，*rScore* の分布に偏りがあったりする場合は，平均値を用いることは不適切である恐れがある．このような場合は，平均値ではなく中央値や箱ひげ図を用いた分析を行う方が望ましい．また，リファクタリングによって *rScore* が上がるケースと下がるケースの両方が存在する場合，値がどのように変化したかという情報を活用することも有用である可能性がある．

本研究では表 4.2 に記載の 5 種類のリファクタリングを実験対象とした．リファクタリングパターンは数多く存在するため，他のパターンで検証することで，新たな傾向を発見したり，今回発見した傾向を否定する例が現れたりする可能性がある．

本提案手法による SBFL スコアの計測結果は，テストスイートとミュータント生成器に影響を受けるため，それらの要素が変化することで今回の実験結果と異なる結果が得られる可能性がある．本研究では，それらの要素が SBFL スコアにどのような影響を与えるかを評価できていない．評価を行う場合は次の方法が考えられる．テストスイートについては，SBFL の精度を高めるためのテストケースを生成する既存研究 [56,57] を活用することで，テストスイートの生成方法が結果にどのような違いをもたらすか検証する方法が考えられる．また，ミュータント生成器については，本研究で用いた 11 種類のミューテーション演算子に加え，別のミューテーション演算子を適用することで，ミュータント生成器による結果の違いを検証する方法が考えられる．特に，今回用いたミューテーション演算子はいずれも，単一の文や式を変更する典型的かつ簡易なものであったが，カプセル化や継承のようなオブジェクト指向特有の機能を変更するミューテーション演算子を適用した場合は，異なる傾向が得られる可能性がある．なお，このようなミューテーション演算子を用いる場合は，実験対象とするプログラムとしても，今回用いたメソッド単体ではなく，継承関係などを含めたある程度の規模のものを用意する必要がある．

なお，リファクタリングはプログラムの内部構造の改善によってプログラムの保守性を高めるための技術であるが，本実験結果では，保守性を高めるためのリファクタリングが SBFL 適合性の低下に繋がるケースが存在した．反対に，SBFL 適合性を高めるためのプログラム変換が，かえって保守性を低下させることにつながる可能性もある．本研究ではプログラム構造の違いによる SBFL スコアの違いについてのみ着目したため，他の品質特性や副特性への影響についての評価ができていない．そのため，SBFL 適合性と他の品質特性との関係を明らかにすることは今後の重要な課題である．

4.7 まとめと今後の課題

本研究では，プログラム自体が SBFL にどの程度適しているかという特性を持っていると考え，その特性を SBFL 適合性として提案した．また，SBFL 適合性を評価する一つの指標を SBFL スコア

とし，ミューテーションテストを活用した SBFL スコアの計測手法を提案した．リファクタリングを題材に，プログラム構造の違いによる SBFL スコアの違いを確認した結果，同じ機能，同じテストスイートであっても，プログラム構造の違いにより SBFL スコアが変化することを確認した．また，SBFL スコアを向上させるプログラム構造の特徴を発見した．

なお，今回用いたリファクタリングパターンはいずれも簡易な変換であったため，発見されたプログラム構造の特徴も直感的に理解のしやすい内容であった．また，リファクタリングは人がソースコードを読むことを前提としたプログラム変換であり，本実験ではリファクタリング前後のプログラムに対する SBFL 適合性の計測結果を比較しているため，今回得られたプログラム構造の特徴は，SBFL にとって相対的な評価結果に過ぎず，真に SBFL 適合性の高いプログラム構造の特徴を捉えられていない可能性がある．今後は，SBFL という手法，すなわち機械が処理することを前提に，より複雑なプログラム構造を考慮することで，SBFL に適する新しいプログラム構造の発見を目指したいと考えている．

また，本研究では SBFL スコアという評価指標を用いることで，SBFL 適合性の高いプログラム構造を発見することができたが，SBFL スコアは使用できる場面が限定的である．具体的には，SBFL スコアの計測にはすべてのテストケースを通過するプログラムが必要であるため，SBFL を用いて欠陥限局を行う前に SBFL 結果の信頼度を把握したいという要求に応えることができない．また，本実験における SBFL スコアの計測対象が非常に少なく，SBFL 適合性が高いと判断するための SBFL スコアのしきい値も不明瞭である．より実用的な場面で SBFL 適合性を活用するためには，実在するプログラムの計測による実験データの収集，SBFL スコア計測手法の前提条件の緩和，あるいは SBFL スコア以外の新たな評価指標についての検討など，さらなる研究が求められる．

加えて，4.6 節で述べたとおり，SBFL 適合性は保守性の一つの観点ではあるものの，他の保守性と相反する可能性がある．そのため，SBFL 適合性を向上させるプログラム変換内容は必ずしも人にとって分かりやすいものとは限らない．そこで今後は，SBFL 適合性の高いプログラム構造と，可読性の高いプログラム構造の関係についての調査や，元のプログラムから SBFL 適合性の高いプログラムへの自動変換手法の提案にも取り組みたい．たとえば，仮に可読性が低くても SBFL 適合性が高いプログラム構造であれば，SBFL を実行する前にそのような構造へ一時的に自動変換することで，元のプログラムの保守性は保ちつつ，欠陥限局の精度だけを向上させることができると考えられる．あるいは，プログラム自動変換前後の文の対応関係を追跡することができれば，元のプログラムは一切書き換えずに，SBFL 実行前の内部処理として変換を行い，欠陥箇所を効率的に特定する方法も考えられる．いずれの場合も，ソースコードは人が読むこともあれば SBFL のような機械が処理することもあるという点を考慮し，ソースコードを取り扱う人または機械にとって最適なプログラム構造を選択していくことが，保守の効率化に繋がる第一歩となると考えられる．

第 5 章

むすび

5.1 まとめ

本研究では，ソフトウェア保守の効率化を目的として，ソースコードの内部構造に着目して保守性を計測または改善する次の三つの研究を行った．

1. メトリクス計測の前処理としてのソースコード簡略化
2. プログラム文の並び替えによるソースコードの可読性向上
3. ソースコードに対する欠陥限局の適合性計測

1. については，従来のメトリクスを用いて理解性の低いモジュールを発見しようとする際，繰り返し記述される制御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．被験者実験の結果，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．更に，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．

2. については，ソースコードの可読性を向上することを目的に，変数の定義と参照の間の距離に着目して，ソースコード中の文を並べ替える手法を提案した．ある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ところ，約 3,700 の並べ替え可能なメソッドに対し，215 のメソッドを並べ替えることができた．また，並べ替えが行われたメソッドのうち，20 個のメソッドを対象に，オリジナルのソースコードと提案手法適用後のソースコードの可読性を 44 名の被験者が比較した．その結果，提案手法を適用することで多くのメソッドの可読性が向上することが確認できた．更に，可読性に影響を与える文の並びには，変数の定義と参照の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．

3. については，欠陥限局手法の一つである SBFL に着目し，ソースコード自体が SBFL にどの程度適しているかという特性を持っていると考え，その特性を SBFL 適合性として提案した．また，SBFL 適合性を評価する一つの指標を SBFL スコアとし，ミューテーションテストを活用した SBFL

スコアの計測手法を提案した．リファクタリングを題材に，ソースコード構造の違いによる SBFL スコアの違いを確認した結果，同じ機能，同じテストスイートであっても，ソースコード構造の違いにより SBFL スコアが変化することを確認した．また，SBFL スコアを向上させるソースコード構造の特徴を発見した．

5.2　今後の研究方針

今後は，本論文で述べた研究成果を応用し，実際のソフトウェア保守作業の効率化に貢献したいと考えている．

1. については，繰り返し構造の折りたたみを手法を応用したソフトウェア保守支援を行いたい．たとえば 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し，ソースコードの理解支援に貢献することを考えている．

2. については，本研究で用いた提案手法のプロトタイプに実装されていない機能を実装することで，並べ替えを行ってもプログラムの振る舞いを保つよう改善し，実用的なツールとしたい．また，対話的な文の並べ替え支援環境を構築することで，開発者がプログラムを読んで理解しようとする場面や，可読性向上のためのリファクタリングを検討する場面において，開発者が望む並べ替え結果を提示できると考えている．

3. については，元のプログラムから SBFL 適合性の高いプログラムへの自動変換手法の提案に取り組みたい．本研究における実験では，一般的にプログラムの品質を向上すると考えられているリファクタリング操作が，SBFL 適合性を低下させてしまう可能性が見つかっている．そのため，SBFL 適合性を向上させるプログラムの変換内容は，必ずしも人にとって分かりやすいものとは限らない．SBFL による欠陥限局は人ではなく機械が行うことを踏まえると，SBFL に適したプログラム変換もまた機械が行うべきである．SBFL 実行前に，SBFL 適合性を向上させるプログラム変換を自動で行うことで，SBFL の精度を向上できると考えている．

参考文献

[1] 増井和也, 馬場辰男, 松永真, 弘中茂樹. ソフトウェア保守開発〜ISO14764 による〜. ソフトリサーチセンター, 2007.

[2] Stephen W. L. Yip and Tom Lam. A software maintenance survey. In *Proceedings of the 1st Asia-Pacific Software Engineering Conference*, pp. 70–79, 1994.

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

[4] ISO/IEC/IEEE 24765:2017 Systems and software engineering - Vocabulary. Standard, International Organization for Standardization, 2017.

[5] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pp. 73–87, 2000.

[6] ISO/IEC 14764:2006 Software engineering - Software life cycle processes - Maintenance. Standard, International Organization for Standardization, 2006.

[7] JIS X 0161:2008 ソフトウェア技術—ソフトウェアライフサイクルプロセス—保守 Software engineering – Software life cycle processes – Maintenance. Standard, 一般財団法人 日本規格協会, 2008.

[8] 室谷隆. 共通フレーム 2013 概説. *SEC Journal*, Vol. 9, No. 1, pp. 19–22, 2013.

[9] Alain Abran and Hong Nguyenkim. Analysis of maintenance work categories through measurement. In *Proceedings of the 1991 Conference on Software Maintenance*, pp. 104–113, 1991.

[10] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. 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, January 2001.

[11] ISO/IEC 25010:2011, Systems and software engineering — Systems and software quality requirements and evaluation (SQuaRE) — System and software quality models. Standard, International Organization for Standardization, 2011.

[12] JIS X 25010:2013 システム及びソフトウェア製品の品質要求及び評価 (SQuaRE) —システム及びソフトウェア品質モデル Systems and software engineering – Systems and software quality requirements and evaluation (SQuaRE) – System and software quality models. Standard, 一般財団法人 日本規格協会, 2013.

[13] Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software*, Vol. 82, No. 6, pp. 981–992, 2009.

[14] Mari Matinlassi and Eila Ovaska. The impact of maintainability on component-based software systems. In *Proceedings of the 29th Conference on EUROMICRO*, pp. 25–32, September 2003.

[15] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability factors? In *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 306–315, 2012.

[16] Rajiv Banker, Srikant Datar, and Dani Zweig. Software complexity and maintainability. In *Proceedings of the 10th International Conference on Information Systems*, pp. 247–255, January 1989.

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

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

[19] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering*, pp. 30–38, March 2001.

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

[21] 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, November 2008.

[22] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. CRANE: Failure prediction, change analysis and test prioritization in practice – experiences from windows. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation*, pp. 357–366, 2011.

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

[24] 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.

[25] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verifica-

tion. *IBM Systems Journal*, Vol. 41, No. 1, pp. 4–12, 2002.

[26] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.370.9611`, 2013.

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

[28] 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.

[29] 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.

[30] 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, March 2011.

[31] 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.

[32] Claire Le Goues and Westley Weimer. Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering*, Vol. 38, No. 1, pp. 175–190, January 2012.

[33] 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.

[34] Ahmad Jbara, Adam Matan, and Dror G. Feitelson. High-MCC functions in the Linux kernel. In *Proceedings of the 34th International Conference on Program Comprehension*, June 2012.

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

[36] Benjamin Biegel, Fabian Beck, Willi Hornig, and Stephan 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.

[37] 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.

[38] Phillip 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.

[39] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactor-

ing opportunities. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, pp. 119–128, March 2009.

[40] 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, June 2000.

[41] 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 the 9th IEEE International Software Metrics Symposium*, pp. 351–360, September 2003.

[42] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707–740, 2016.

[43] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, Vol. 37, No. 5, pp. 649–678, 2011.

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

[45] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro 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.

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

[47] 村上寛明, 堀田圭佑, 肥後芳樹, 井垣宏, 楠本真二. ソースコード中の繰り返し部分に着目したコードクローン検出ツールの実装と評価. 情報処理学会論文誌, Vol. 54, No. 2, pp. 845–856, February 2013.

[48] Hiroaki Murakami, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Folding repeated instructions for improving token-based code clone detection. In *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 64–73, September 2012.

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

[50] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 1st edition, 2009.

[51] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pp. 467–477, 2002.

[52] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 595–604, 2002.

[53] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization

for Java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pp. 528–550, 2005.

[54] Andréia da Silva Meyer, Antonio Augusto Franco Garcia, Anete Pereira de Souza, and Cláudio Lopes de Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, Vol. 27, No. 1, pp. 83–91, 2004.

[55] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pp. 89–98, 2007.

[56] Tao Wang and Abhik Roychoudhury. Automated path generation for software fault localization. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pp. 347–351, 2005.

[57] Han-Lin Lu, Ruizhi Gao, Shih-Kun Huang, and W. Eric Wong. Spectrum-base fault localization by exploiting the failure path. In *Proceedings of the 2016 International Computer Symposium*, pp. 252–257, 2016.

[58] Rudolf Ramler, Thomas Wetzlmaier, and Claus Klammer. An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the 32nd Annual ACM Symposium on Applied Computing*, pp. 1401–1408, 2017.

[59] Sun-Woo Kim, John Clark, and John McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. *Software Testing, Verification and Reliability*, Vol. 11, pp. 207–225, December 2001.

[60] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pp. 352–363, 2002.

[61] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava: A mutation system for Java. In *Proceedings of the 28th International Conference on Software Engineering*, pp. 827–830, 2006.

[62] René Just. The major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*, pp. 433–436, 2014.

[63] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 449–452, 2016.

[64] Dana Halabi and Adnan Shaout. Mutation testing tools for Java programs – a survey. *International Journal of Computer Science and Engineering*, Vol. 5, pp. 11–22, June 2016.

[65] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 7th IEEE International*

Conference on Software Testing, Verification and Validation, pp. 153–162, 2014.

[66] Mike Papadakis and Yves Le Traon. Metallaxis-fl: Mutation-based fault localization. *Software Testing, Verification and Reliability*, Vol. 25, No. 5–7, pp. 605–628, August 2015.

[67] Dustin Boswell and Trevor Foucher. *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. O'Reilly Media, 2011.

[68] Henry Coles. PIT. `https://pitest.org/`, accessed August, 2020.