特別研究報告

題目

ソースコード差分検出を用いた探索的手法による impure リファクタリングの検出

指導教員
井上 克郎 教授

報告者
堤 祥吾

平成 28 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成27年度 特別研究報告

ソースコード差分検出を用いた探索的手法によるimpureリファクタリングの検出

堤 祥吾

内容梗概

リファクタリングとは，ソフトウェアの外部的振る舞いを保ったままで，内部の構造を改善していく作業をいう．このリファクタリングがソースコードの品質に与える影響について，現在関心が持たれている．リファクタリングによる影響を調査するために，ソースコードの変更において適用されたリファクタリングを自動的に検出できるようになることが有用である．現状として，リファクタリングの検出は，バージョン管理システムのソースコード変更履歴を解析することで行われている．しかし，ソースコードにリファクタリングが機能追加・変更とともに適用された場合や，複数のリファクタリングがまとめて適用された場合に，リファクタリングを検出することが困難になっている．このようなリファクタリングはimpureリファクタリングと呼ばれている．

林らは，探索的手法を用いて複数のリファクタリングを検出可能にする研究を行った．林らの手法ではまず，2コミット間の変更に対して適用されたリファクタリングの候補を列挙する．そして，旧版に対して探索的にリファクタリングを適用していき，新版と一致したときのリファクタリング操作列を検出リファクタリングとする．しかし林らの方法では，リファクタリングと同時に非リファクタリング変更が加えられた場合にリファクタリングを検出することができない．そこで，本研究では林らの探索的手法に加えて，ソースコード差分検出とテストの実行によってimpureリファクタリングも検出する手法を考案した．本研究ではJavaのソースコードを対象とするため，オープンソースプロジェクトであるThe Apache Xercesのリポジトリからあるコミットを選択し，提案手法を適用した．

主な用語

リファクタリング

ソースコード差分検出

探索的手法

目 次

1 まえがき **3**

2 リファクタリング **4**

2.1 リファクタリングのメリット 4

2.2 リファクタリングパターン 4

2.2.1 メソッド抽出 5

2.2.2 メソッド引き上げ 5

2.2.3 impure リファクタリング 7

2.3 リファクタリング検出手法 7

2.3.1 検出規則を用いた手法 7

2.3.2 コードクローン検出を用いた手法 9

2.3.3 探索を用いた検出手法 9

3 提案手法 **12**

3.1 探索的手法によるリファクタリング検出 13

3.1.1 評価関数による評価値の算出 13

3.1.2 重複状態での探索打ち切り 14

3.2 探索後の状態の動作確認 14

3.3 非リファクタリング差分検出 15

3.3.1 ソースコードをトークン列に変換 15

3.3.2 メンバのマッチング 16

3.3.3 レーベンシュタイン距離を用いたトークン列の差分検出 16

4 適用実験 **19**

4.1 結果 19

4.2 考察 21

5 まとめ **23**

謝辞 **24**

参考文献 **25**

付録 **27**

1　まえがき

リファクタリングとは，ソフトウェアの外部的振る舞いを保ったままで，内部の構造を改善していく作業をいう [5]．ソースコードが設計の欠陥や度重なる変更，追加により複雑化してしまった場合に，リファクタリングを適用することで，外部から見た振る舞いを変更せずにソースコードを修正することができる．リファクタリングは Fowler によっていくつかのパターン [5] に分類され，多くの研究で各パターンを 1 つのリファクタリングとみなしている．

現在，リファクタリングがソースコードの品質に与える影響について関心が持たれている．[14][15] それには，バージョン管理システムのソースコード変更履歴からリファクタリングの検出を行うことが必要である．調査において効率的にリファクタリングを検出するためには，自動的にリファクタリング検出を行うことができるツールが必要である．そこで，リファクタリング検出手法について多くの研究がなされている [13]．

林らは，探索的手法を用いることで，複数のリファクタリングが同時に適用されていても検出する手法を提案した．林らの手法では，ある 2 コミット間で適用されたリファクタリングを検出したいとするとき，まず適用された可能性のあるリファクタリングを列挙する．列挙したリファクタリング操作を，探索的に旧版コードに適用していき，新版コードと一致した場合，得られたリファクタリング操作列を検出結果とする．

しかし林らの手法 [6] では，リファクタリング以外の操作が適用された場合に検出結果を出すことはできない．そこで本研究では，林らの手法に加え，ソースコードの差分検出とテストを用いることで，そのようなリファクタリングを検出することを可能とする．

具体的には，林らの手法における探索において，新版コードに一致するリファクタリング操作列が見つかった場合は林らの手法同様にそれを検出結果とする．一定回数の探索で見つからなかった場合，もっとも新版コードに近づいた場合に，その状態に至るまでに適用したリファクタリングをリファクタリング操作列とし，探索後のソースコードと新版コードとの差分を非リファクタリング操作列とする．また，テストを用いることで動作の同一性を保証する．

本研究では，Java のソースコードを対象とし，提案手法を Eclipse プラグインとして実装した．また，オープンソースプロジェクトである The Apache Xerces[1] に対して提案手法を適用し，複数のリファクタリングやリファクタリング以外の変更が適用された impure リファクタリングを検出できることを確認した．

以下，2 章では本研究の前提知識となるリファクタリングを説明し，3 章では，提案手法を説明する．4 章では適用結果を示し，5 章ではまとめと今後の課題について述べる．

2　リファクタリング

この章では，リファクタリングの説明と例を紹介し，impure リファクタリングについて説明する．そして，既存のリファクタリング検出手法の問題点と，先行研究である探索的手法を用いたリファクタリング検出について述べる．

リファクタリングとは，ソフトウェアの外部的振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させることであると Fowler の著書で定義されている [5]．ソースコードが設計の欠陥や度重なる変更，追加により複雑化してしまった場合に，リファクタリングを適用することで，外部から見た振る舞いを変更せずにソースコードを修正することができる．リファクタリングによって，ソースコードの設計を後から向上させることができるのである．

2.1　リファクタリングのメリット

Fowler は著書 [5] でリファクタリングのメリットを 3 つ説明している．

- リファクタリングはソフトウェアを理解しやすくする
- リファクタリングはバグを見つけ出す
- リファクタリングでより速くプログラミングできる

リファクタリングはコードを読みやすい形に修正する作業を言う．そのため，リファクタリングによってコードを他者が理解しやすくなるのは明白である．また，リファクタリングを行う際にコードを理解し，その理解をリファクタリングによってコードに反映することで，自身によるコードの理解も深まる．リファクタリングによってコードが単純で理解しやすくなることで，バグの発見やプログラミング速度の上昇にもつながる．

このような理由により，リファクタリングが行われている．

2.2　リファクタリングパターン

リファクタリングは Fowler によっていくつかのパターン [5] に分類され，多くの研究で各パターンを 1 つのリファクタリングとみなしている．ツールや手動でリファクタリングを行う場合，パターン単位でリファクタリングが行われている．

リファクタリングパターンの例として，以下ではメソッド抽出とメソッド引き上げを紹介する．

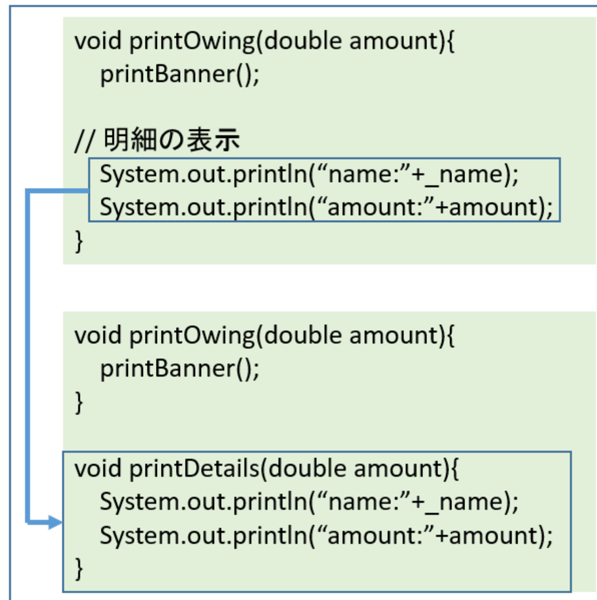

図 1: メソッド抽出の例

2.2.1　メソッド抽出

メソッド抽出は，長い処理やひとまとまりの処理を取り出して，新しいメソッドを作成するリファクタリングである．メソッド抽出を行うことの利点として，単一メソッドの長さを短くすることで，ソースコードの見通しをよくする点や，メソッド化されたソースコードを再利用しやすくなる点が挙げられる．

図 1 は，支払いの明細を表示する “printOwing” メソッドにメソッド抽出リファクタリングを適用した例である．ひとまとまりの処理である明細の表示部分を新しく作成した “printDetails” メソッドに移すことで，メソッドの機能を細かく分割している．また，処理自体に名前がつくことで “printOwing” メソッドの概要を素早く把握することができるようになる．

2.2.2　メソッド引き上げ

メソッド引き上げは，サブクラスのメソッドをスーパークラスに移動するリファクタリングである．

図 2 は顧客を表す “Customer” クラスとそのサブクラスに対し，メソッド引き上げを適用した例である．両方のサブクラスに税込み価格を計算する “culcTax” メソッドが存在し，どちらも処理内容は同じソースであるため，スーパークラスである “Customer” クラスに移動させている．これにより，コードの可読性を上げ，バグ発生の可能性を下げている．

複数のサブクラスに同じメソッドが存在する場合，あるメソッドに対して行った変更がほ

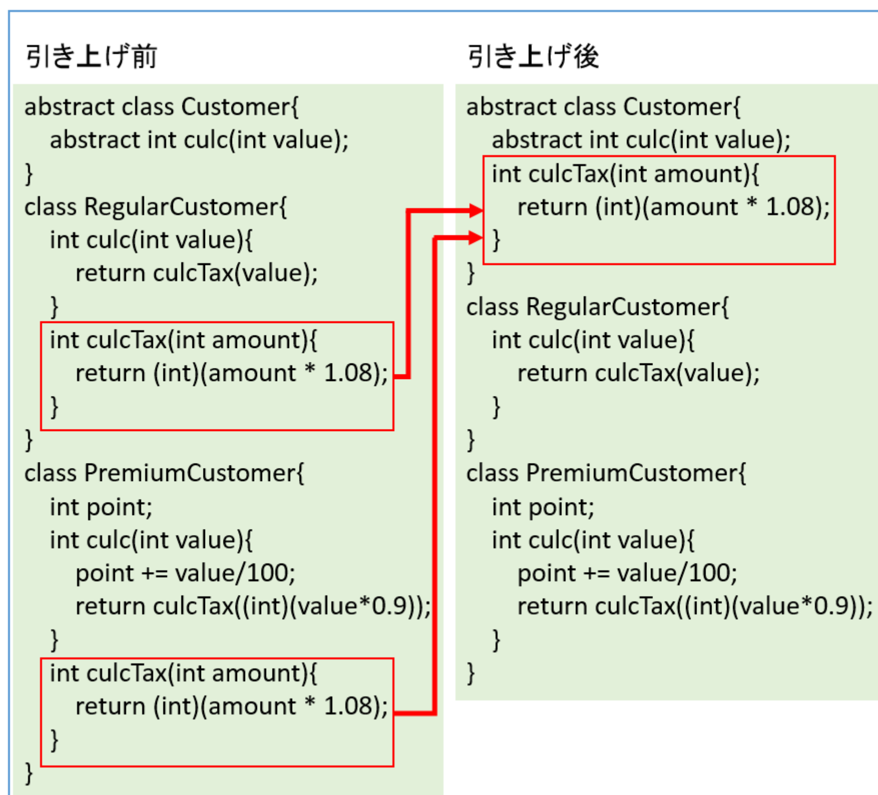

図 2: メソッド引き上げの例

かのメソッドに対しても同様に行われる保証はない．もし変更漏れがあればバグとなりうる．また，同じメソッドが別の場所に存在することで，コードの可読性も減少してしまう．メソッド引き上げは，これらの問題を解消することが可能である．

2.2.3 impure リファクタリング

版管理システムにおいて，一度のコミットで単一のリファクタリングのみが適用されている場合もあれば，そうでない場合もある．つまり，ソースコードに複数のリファクタリングが組み合わせて適用されている場合や，バグ修正や機能の追加など，リファクタリング以外の変更 (以下，非リファクタリング変更) が同時に適用されている場合である．Gorg らは，このような変更を impure リファクタリングと呼んでいる [4]．impure リファクタリングでは，かならずしも外部的振る舞いが保存されるわけではない．

図 3 は Apache Ant のリポジトリにおける，あるコミットの変更の抜粋である．処理の詳細は割愛するが，コンストラクタ内でファイル名の正規化と確認が行われており，メソッド抽出と非リファクタリング変更が混在している．青枠で表された部分はメソッド抽出であり，fileName を生成する一連の処理を “normalizeFileName” メソッドとして抽出している．また，下の赤い枠内のコードは上の赤い枠内のコードに非リファクタリング変更が加えられたコードである．

このように，リファクタリングと非リファクタリング変更が同時に適用されている変更や，複数のリファクタリングが適用されている変更を impure リファクタリングと言う．

2.3 リファクタリング検出手法

リファクタリングがソースコードの品質に及ぼす影響について，現在関心が持たれている．リファクタリングが及ぼす影響を調べるためには，実際に行われているリファクタリングを検出し，どのような変化があったかを調査することが必要である．

現状，リファクタリングを検出するために，バージョン管理システムのソースコード変更履歴の解析が行われているが，大規模なプロジェクトでのリファクタリングを目視で検出することは現実的ではない．そこで，リファクタリングの実施を自動的に検出する手法が数多く提案されている [13]．

2.3.1 検出規則を用いた手法

Prete らの手法では，版間の差分があらかじめ定められたパターン (検出規則) に一致する場合はリファクタリングが実施されたとする，検出規則を用いた手法をとっている [8]．以下に例としてメソッド引き上げの検出規則を示す [10]．

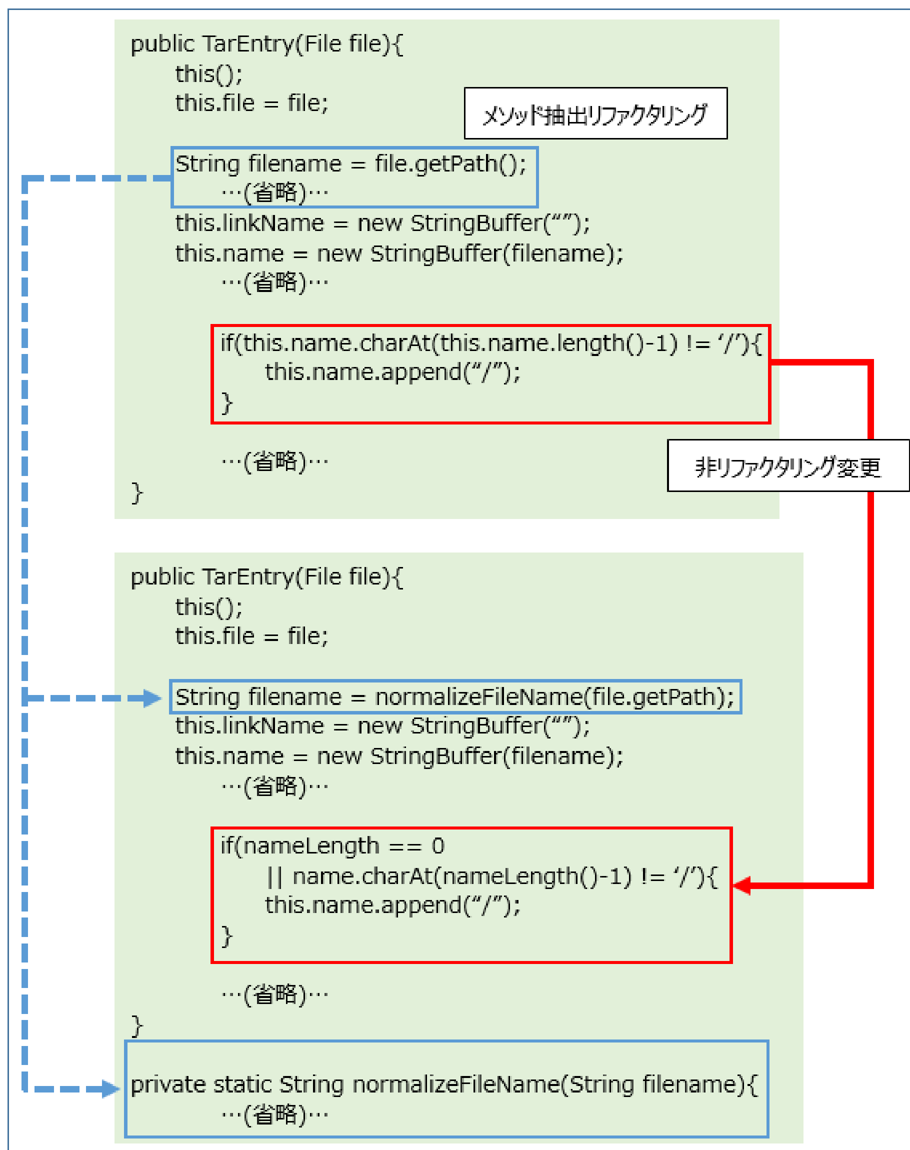

図 3: impure リファクタリングの例

メソッド引き上げ (pull up method) の検出規則

```
move_method(fShortName, tChildFullName, tParentFullName)
∧ before_subtype(tParentFullName, tChildFullName)
⟶ pull_up_method(fShortName, tChildFullName, tParentFullName)
```

上の例によると，メソッド引き上げは tChildFullName クラスにある fShortName メソッドが tParentFullName クラスに移動されたという条件 (`move_method`) と，tChildFullName クラスが tParentFullName のサブクラスであるという条件 (`before_subtype`) を満たしているリファクタリングである，ということを示している．このように，あらかじめ定義されている基本的なソースコード変更パターンを検出し，それら基本的パターンの組み合わせがリファクタリングパターンと一致している場合に，リファクタリングが検出されたとする．検出規則を定義しておけばそれに応じたリファクタリングを検出することができる．一方，検出規則の検討が不十分である場合に検出精度が低下する欠点がある．また，非リファクタリング変更が存在する impure リファクタリングの検出には向かない．

2.3.2 コードクローン検出を用いた手法

Weissgerber らは，版間の変更において，類似したコード片がどこに移動，抽出されたかを追跡することでリファクタリングを検出した [11].

追跡にはコードクローン検出ツール [3][9] を用いるため，移動，抽出されたコード片が完全に一致していなくとも追跡することができる．非リファクタリング変更が存在してもリファクタリング検出が行える可能性があるが，複数の種類の変更やリファクタリングが適用されている場合の検出には向かない．

2.3.3 探索を用いた検出手法

上記，検出規則，コードクローン検出を用いた手法は，同一箇所に複数のリファクタリングが適用されている場合に，リファクタリングを検出することができない．しかし，探索を用いた手法により，そのような場合でもリファクタリングを検出することが可能となる．

林らは，探索的手法を用いてリファクタリングを検出する手法を提案している [6]．提案手法では，プログラムを状態，リファクタリング操作を状態遷移とみなし，版間の変更と一致するリファクタリング操作列を探索する．これにより，複数のリファクタリングが同時に適用されている場合においてもリファクタリング操作列を検出することを可能としている．

以下，林らの論文をもとに手法を説明する．

概要

林らは，プログラム P_{old}，P_{new} 間に行われたリファクタリング操作の発見を，以下の探索問題として表現している．

- 状態：プログラム
- 初期状態：$P_{old}(= n_0)$
- 目標状態：$G = \{P_{new}(= n_m)\}$
- オペレータ：リファクタリング操作
- 経路コスト：それまでに行ったリファクタリング操作の回数

この手法では，経路探索として A*探索 [12] が用いられている．探索中の状態 n を含むコストは，評価関数 $f(n) = g(n) + h(n)$ で与えられ，$g(n)$ は経路コスト，$h(n)$ は n から目標状態 $(= n_m)$ までの予想コストを経験的に与えるヒューリスティックである．この $h(n)$ が，探索の効率に直結するため，ヒューリスティックを工夫する必要がある．

探索の流れ

探索の各状態で，以下を繰り返す．

1. n，n_m 間の差分をもとに $h(n)$ を計算する．
2. 差分をもとにリファクタリング操作候補を選出し，順位付けを行う．
3. 順位をもとに，n にリファクタリング操作を適用していき，得たソースコード群を新たに n とする．

これを，n と n_m との差分がなくなったときの n か，一定回数繰り返した後で最も $h(n)$ が小さい n を取り出し，その状態に至るまでに適用したリファクタリング操作列を結果として得る．

ヒューリスティックの計算

この手法で使われたヒューリスティックは以下の式で表される．

$$h(n, o) = h'(n)/\alpha(o)$$

ここで $h'(n)$ は状態の差分をもとに計算した距離のヒューリスティックである．また，$\alpha : O \to (0, 1]$ は選出したリファクタリング操作候補の評価値であり，その候補が実際に行われた可能性が高いほど大きい値をとる．そのため $h(n, o)$ は，評価の低いリファクタリングを適用する場合は，ソースコードの差分から考える距離 $h'(n)$ よりも大きな値をとるようになっている．

手法の問題点

林らの手法では，非リファクタリング変更が版間に適用されている場合は完全な操作列の検出が不可能である．本研究では，林らの手法に加え，ソースコード差分検出技術を用いることで，非リファクタリング変更が適用されている場合でも操作列を検出することを可能にする．

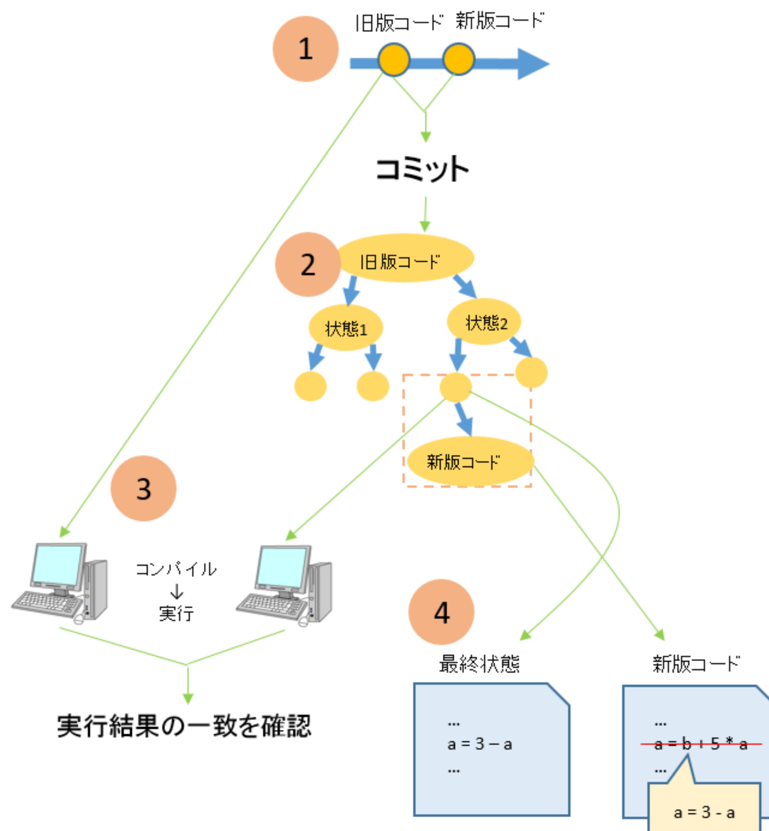

図 4: 提案手法の概要

3　提案手法

本研究では，探索的手法によるリファクタリング操作列検出とソースコード差分検出技術の組み合わせにより，impure リファクタリングを検出する手法を提案する．

提案手法の概要は図 4 の通りである．

1. 対象プロジェクトからあるコミット (旧版コード) とそれより一つ新しいコミット (新版コード) を取り出す
2. 探索的手法により検出したリファクタリング操作列を適用し，新版コードに近づいた旧版コード (探索後コードとする) を得る
3. リファクタリング操作列を適用した後正常に実行できるか動作確認を行う
4. 探索後コードと新版コードの差分を検出してそれを非リファクタリング操作列とし，探索後コードに適用する

上記手法を適用後，リファクタリング操作列と非リファクタリング差分を結果として得る．

3.1 探索的手法によるリファクタリング検出

探索の手法は林らの手法 [6] を参考に実装を行った．検出の流れは以下の通りである．

1. 旧版コード (初期状態) を，評価値 (3.1.1 にて説明) とともに優先度つきキューに入れる
2. キュー内で最も評価値の優れた状態 (s_i とする) を取り出す
3. s_i から新版コードへの変更で，適用されたと考えられるリファクタリングを列挙する (δ_0,δ_1,...とする)
4. s_i に対して各 δ_0,δ_1,... を適用し，新たな状態 $\delta_0(s_i)$,$\delta_1(s_1)$,... を得た後，各評価値を計算し，キューに入れる
5. 終了条件を満たしていなければ 2 に戻る．満たしていれば，探索中最も評価値の優れていた状態を探索結果として得る

終了条件とは，

- キューが空になる
- 探索中のコードと新版コードが一致する
- 一定時間 (本研究では 600 秒としている) が経過する

のいずれかである．

探索的手法に基づくリファクタリング検出の段階は以上であり，次の段階で，上記手法で得られた最終状態のテストを行う．

3.1.1 評価関数による評価値の算出

探索中に取り出される状態 (以下，探索状態) の優先順位をつけるために評価関数を用いて評価値を計算している．本研究では，目的状態に早くたどり着くため，3.3.3 で説明するレーベンシュタイン距離 [7] を利用し，評価値を算出している．

評価値の求め方を説明する．レーベンシュタイン距離は名前の一致するメンバごと，トークン単位で求めている．ここで，探索状態のメンバ (A_1, A_2,..., A_n) とマッチング (後述) する目的状態のメンバを (B_1, B_2,..., B_3) とする．各 A_i と $B_i (i = 1, 2,..., n)$ とのレーベンシュタイン距離を d_i，A_i, B_i のトークン数をそれぞれ a_i, b_i とする．このとき，探索状態の評価値は，

$$\frac{\sum_{i=1}^{n} d_i}{\sum_{i=1}^{n} max(a_i, b_i)}$$

として表される．分母である総トークン数は分子である差分より大きくなるため，評価値は $[0, 1]$ の値をとる．

3.1.2 重複状態での探索打ち切り

たとえば，メソッド A，B があったとして，A にメソッド引き上げを適用した後に B にメソッド引き上げを適用した場合と，B にメソッド引き上げを適用した後に A にメソッド引き上げを適用した場合に，同じ状態にたどり着いてしまう場合がある．同じ状態に対して探索を実行するのは無駄であるので，2 回目以降で同じ状態にたどり着いた場合，そこで探索を打ち切ることが必要である．

本研究では，ソースコードのハッシュ値と評価値両方が一致した場合に探索打ち切りを行うことにした．評価値の求め方は上で説明した通りなので，ハッシュ値の求め方について説明を行う．探索状態のハッシュ値を求めるために，プロジェクト全体のソースコードを文字列として得た後，Java の String クラスの hashCode メソッドと同じアルゴリズムを用いることでハッシュ値を得た．

文字列の各文字を $(s_0, s_1,..., s_{n-1})$ で表すとき hashCode の値は，

$$31^{n-1}s_0 + 31^{n-2}s_1 + ... + s_{n-1}$$

で表される．Java で標準的に用いられており [2] ハッシュの衝突が少ないアルゴリズムのため，この手法を用いた．

3.2 探索後の状態の動作確認

探索によってリファクタリング操作列が適用されたソースコードを得る．ここで，探索後の状態 (探索後コード) の動作と，リファクタリング操作列を適用する前の状態 (旧版コード) の動作が一致するかどうかを確認する．リファクタリングは，正しく適用された場合は動作を変えることはないが，失敗した場合思わぬ動作変更を起こす場合がある．正しく適用されたことを確認するために動作確認を行う．

本研究では，プロジェクト付属のテストケースを利用する．あらかじめ旧版コードのテスト結果を保持しておき，探索後コードとのテスト結果を比較する．結果が一致しない場合はリファクタリング失敗として扱い実行を終了する．結果が一致する場合は非リファクタリング差分検出に移る．

3.3 非リファクタリング差分検出

探索において，すべてのリファクタリングを検出できたものと仮定する．この場合，探索的手法によって得られた最終状態のソースコード (以下，探索後コード) と，新版コードとの差分は非リファクタリング差分となる．

非リファクタリング差分検出手法の流れは以下の通りである．プロジェクトに複数のファイルがある場合は，同一名のファイルごとに以下を適用する．

1. 探索後コードと新版コードをそれぞれトークン列に変換する
2. トークン列同士のレーベンシュタイン距離を算出する

差分を求める際に，空白やコメントを無視したいので，ソースコードはトークン列として扱う．トークン列同士の差分は，一般に文字列に使われるレーベンシュタイン距離算出のアルゴリズムをもとに求めた．

差分を取得した後，各ファイルごとに得られたリファクタリング操作列と非リファクタリング変更について表示を行う．

3.3.1 ソースコードをトークン列に変換

Java ソースコードを，スペースや記号で区切ることでトークン列に変換している．例として，以下のようにソースコードからトークン列への変換が行われる．

元となる Java ソースコード

```
class Example{
    void example(){
        int a = 3 + 4;
    }
}
```

トークン列 (カンマ区切り)

```
[class, Example, {, void, example, (, ), {, int, a, =, 3, +, 4, ;, }, }]
```

差分検出の方法としては，文字単位の差分，トークン単位の差分，行単位の差分が考えられたが，このうちトークン単位の差分を採用した理由としては次の通りである．

ソースコードを変更する際，文字単位で変更を行うことは少ない．さらに，差分検出単位を細かくしてしまうと計算時間が多くかかってしまうという理由もあり，文字単位での差分検出を行わなかった．また，差分検出において空白やコメントなどは無視している．行単位

で差分をとる場合，改行の位置によっては同じ処理が異なるものと判定される可能性がある．このため，本研究ではトークン単位の差分を取得し利用している．

3.3.2 メンバのマッチング

各探索段階においてクラス内のメンバの順序は不定である．本研究の差分検出においてメンバの順序は考慮しないため，マッチングしたメンバごとに差分をとっている．メンバをソートする方法もあるが，計算時間と実装量の問題からこちらの方法を選択した．

マッチングの手法について説明を行う．図 5 は，The Apatch Xerces プロジェクトの DocumentImpl.java ファイルの一部であり，探索中の DocumentImpl.java(以降ファイル A) と探索の最終目標となる DocumentImpl.java(以降ファイル B) でのマッチングを図に表したものである．オレンジ色の線で結ばれたメンバ同士がマッチングしていて，そうでないメンバはマッチング相手となるメンバがなかったということを示している．マッチングしなかったメンバについては，空トークン列とマッチングしたものとして扱う．

マッチング相手の選び方だが，フィールドについては名前の一致，メソッドについては名前と引数型の一致によって行っている．またインナークラスに関して，本来は再帰的に内部のメソッドについてマッチングを行うべきであるが，本研究ではそれを省略し，クラス名でマッチングした後，メソッドと同様にインナークラスごと差分をとるという方法を行っている．

3.3.3 レーベンシュタイン距離を用いたトークン列の差分検出

編集距離の一つであるレーベンシュタイン距離 [7] は，2 つの文字列がどの程度異なるかを示す距離である．1 文字の削除，挿入，置換の 3 つの操作を組み合わせ，一方の文字列からもう一方の文字列に変換するための最小操作回数をいう．

一般には文字列に用いられる距離であるが，本研究では文字をソースコードのトークンに置き換えて考えることで，トークン列同士の距離を算出している．

以下，簡単にレーベンシュタイン距離算出アルゴリズムの説明を行う．図 6 はレーベンシュタイン距離を求める際に用いる表を表している．表の座標を (x, y) で表し，左上の白いマスの 0 を $(0, 0)$ として，左から右へ x 座標が増加し，上から下へ y 座標が増加するものとする．このとき (x, y) は，“a=b” の x 番目までのトークン列と “a=c+b” の y 番目までのトークン列とのレーベンシュタイン距離を表している．たとえば，$(3, 3)$ は “a=b” と “a=c” とのレーベンシュタイン距離を表していて，この場合 b を c に置換するだけでよいので距離は 1 となる．

表を用いて，レーベンシュタイン距離を求めるアルゴリズムの概要を説明する．編集距離

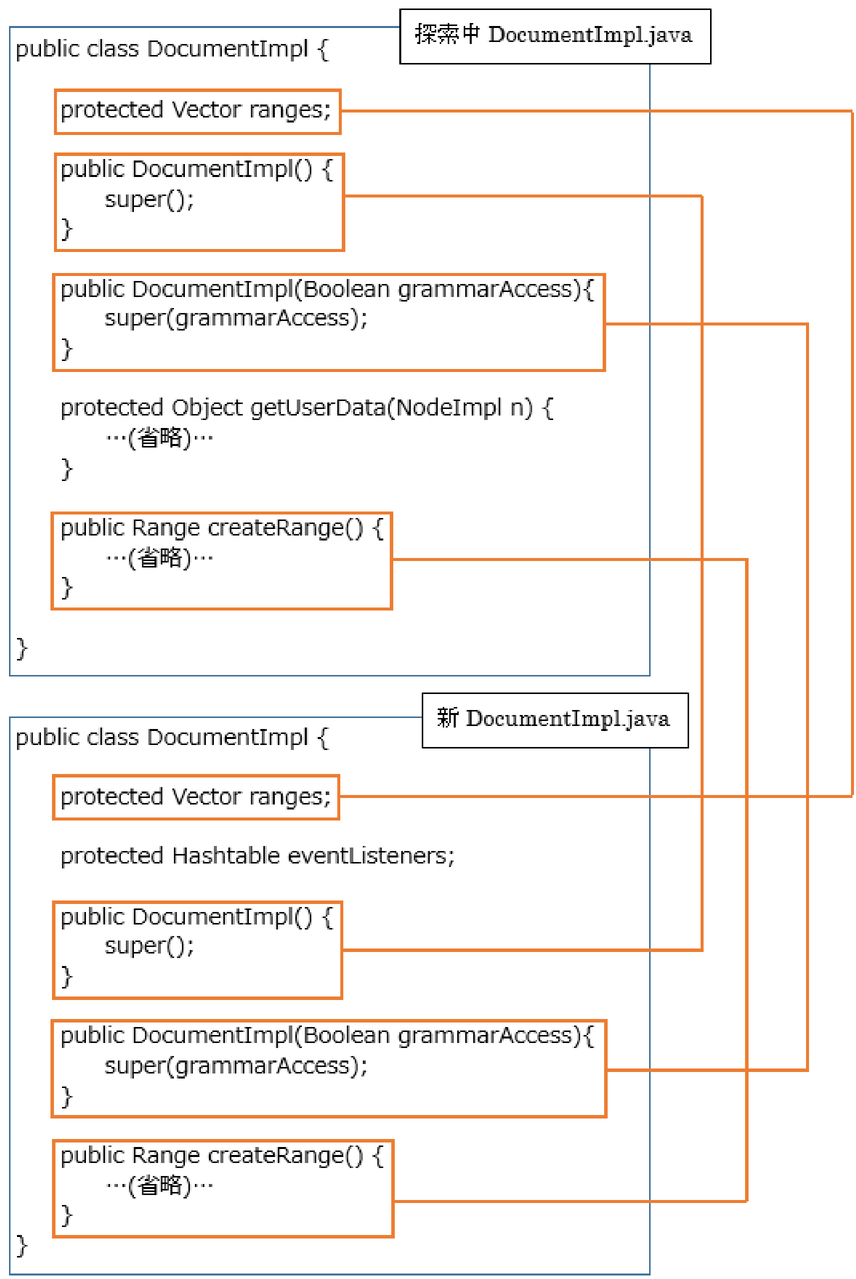

図 5: マッチング

		a	=	b
	0	1	2	3
a	1	0	1	2
=	2	1	0	1
c	3	2	1	1
+	4	3	2	2
b	5	4	3	2

最小編集距離

図 6: レーベンシュタイン距離:アルゴリズム

を求めるための編集は，削除，挿入，置換の 3 つがあり，それぞれが表上の矢印 1，2，3 の遷移に相当する．遷移先の現在のコストよりも，[遷移前のコスト + 編集ごとのコスト (後述)](新コスト) の方が小さい場合，遷移先のコストを新コストで置き換える．

上記操作を繰り返していき，最終的に表の右下に表れる値が最小の編集距離であり，レーベンシュタイン距離となる．

編集ごとのコストについてであるが，削除，挿入の遷移はそれぞれ 1 で，置換については 0 の場合と正コスト (本研究では 2) の場合がある．これは $(x-1, y-1)$ から (x, y) に遷移する場合に，“a=b” の x 番目と “a=c+b” の y 番目が一致する場合については 0 であり，一致しない場合は 2 となる．これは，一致する場合は置換する必要がない (置換コストが 0) ということを表している．

レーベンシュタイン距離のアルゴリズムでは最小編集距離の値が得られるが，表を赤い矢印とは逆方向に辿っていくことで，レーベンシュタイン距離を導出するための編集列を得ることができる．

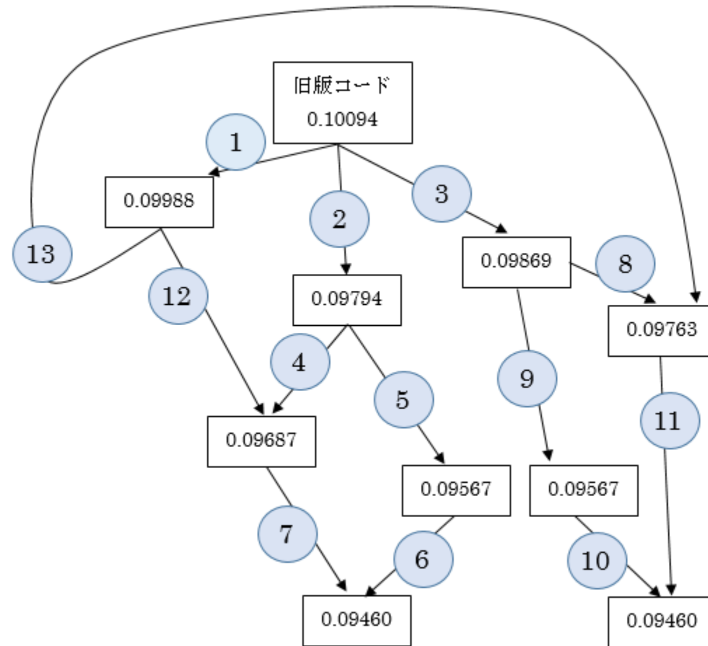

図 7: 探索の状態遷移図

4　適用実験

オープンソースプロジェクトである The Apache Xerces[1] のリポジトリから連続する 2 つのコミットを選択し，本研究の手法を適用した．また，プロジェクトからリファクタリングが含まれるファイルを選択し，コミット間に適用されたリファクタリングを対象として実験を行った．

実験データは以下の通りである．

対象プロジェクト	The Apache Xerces
Commit ID	318022，318023
対象ファイル	DocumentImpl.java，CoreDocumentImpl.java
対象リファクタリング	メソッド引き上げ，フィールド引き上げ

対象とした Commit ID のファイルの内容のうち，リファクタリングの対象となる部分については付録として後半に付けてある．

4.1　結果

図 7 は，実験データに対して探索を行った際の状態遷移を表すグラフである．各四角は探索中にたどり着いた状態を表しており，矢印は状態遷移を表している．矢印上に振ってある

数字の順に探索が行われた．四角内の数字は評価値の値で，値が小さいほど新版コードとの差分が小さいことを示している．各番号が示す遷移で適用されたリファクタリングは以下の表のとおりである．

番号	適用されたリファクタリング	ターゲットメンバ	ターゲットクラス
1	フィールド引き上げ	Hashtable userData	DocumentImpl
2	メソッド引き上げ	setUserData(NodeImpl, Object)	DocumentImpl
3	メソッド引き上げ	getUserData(NodeImpl)	DocumentImpl
4	フィールド引き上げ	Hashtable userData	DocumentImpl
5	メソッド引き上げ	getUserData(NodeImpl)	DocumentImpl
6	フィールド引き上げ	Hashtable userData	DocumentImpl
7	メソッド引き上げ	getUserData(NodeImpl)	DocumentImpl
8	フィールド引き上げ	Hashtable userData	DocumentImpl
9	メソッド引き上げ	setUserData(NodeImpl, Object)	DocumentImpl
10	フィールド引き上げ	Hashtable userData	DocumentImpl
11	メソッド引き上げ	setUserData(NodeImpl, Object)	DocumentImpl
12	メソッド引き上げ	setUserData(NodeImpl, Object)	DocumentImpl
13	メソッド引き上げ	getUserData(NodeImpl)	DocumentImpl

また，探索後に取得した差分で，追加，削除されたトークン数は以下の通りである．以下の変更は非リファクタリング差分であった．

クラス	追加されたトークン数	消去されたトークン数
DocumentImpl	12	0
CoreDocumentImpl	621	47

トークン追加，削除の理由は以下の通りである．

DocumentImpl

理由	追加されたトークン数	消去されたトークン数
import 文の追加	3	0
メソッド内の処理の追加	9	0

CoreDocumentImpl

理由	追加されたトークン数	消去されたトークン数
import 文の追加	3	0
メソッド内の処理の追加	18	0
メソッド内の処理内容の変更	12	47
フィールドの追加	3	0
メソッドの追加	585	0

ソースコードを目視で確認し，実際に適用されていたリファクタリング，非リファクタリング変更が本実験により検出できたことを確認した．

4.2 考察

旧版コードと新版コードとの間で適用されたリファクタリングはすべて検出することができた．また，リファクタリング操作列適用後のコードと新版コードとの差分を検出することで，非リファクタリング変更を検出することができた．実験データに対して本研究の手法により，impure リファクタリングの検出を行うことができた．

探索の状態数

本実験では，対象とするコミットやファイルを限定して探索を行っている．そのため，探索でとった状態数は列挙可能な大きさであったが，対象とするコミットやファイル，候補となるリファクタリングが膨大になった場合は現実的な時間内に十分な空間を探索できない可能性がある．

探索における重複状態取り除き

過去にたどり着いた状態から再度探索を行うのは非効率であるため，重複状態にたどり着いた場合にそれを取り除く処理を行っている．しかし，図 7 の遷移 5，9 と遷移 6，7，10，11 では，評価値が完全に一致しているにも関わらず異なる状態であると判断された．これは，各ファイルのクラスメンバは一致しているが，クラスメンバの順番が異なるのが原因である．

上記状態を同一と判断したい場合，クラスメンバをある規則に従い並べ替え，そのうえでハッシュ値の計算を行うという方法が考えられる．

動作確認を行うタイミング

本研究では，状態遷移におけるリファクタリングの失敗可能性の低さ，テストの時間的コストを考慮し，動作確認を 1 度だけ行うこととしている．本研究で用いた実験データでは，1 度のテストにおよそ 60 秒ほどの時間がかかり，探索における状態遷移がおよそ 2 秒であることに対し 30 倍の時間がかかっている．本研究で最も時間がかかっているのは探索であるため，毎回の探索で動作確認を行った場合に実行時間が 30 倍となってしまう．ただし，テストの実行にかかるコストが十分小さい場合は，状態遷移毎に動作確認を行い，リファクタリング検出の精度向上を図ることが可能であると考えられる．

5　まとめ

本研究では，探索的手法とソースコード差分検出手法を組み合わせることで，impure リファクタリングの検出を行った．既存のリファクタリング検出技術では，複数のリファクタリングが同時に適用されていた場合や，リファクタリング以外の変更が適用されていた場合にリファクタリングを検出することが難しい．本研究の手法を適用することで，そのような impure リファクタリングを含む実験データからリファクタリングを検出することができた．

今後の課題として，複数の改善案が挙げられる．まず，今回適用していないリファクタリング検出機能の追加である．本実験ではメソッド引き上げとフィールド引き上げのみを適用したため，メソッド抽出等，それ以外のリファクタリング機能の追加を行うことが考えられる．また，リファクタリング検出の対象を大きくした場合に，現状では探索空間が大きくなってしまい十分な探索が行えない．本研究では実行時間や探索空間を意識せず手法を実装したが，今後リファクタリング候補の列挙や探索の枝刈りを効率化することも検討していきたい．

謝辞

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 教授には研究について適切な御指導及び御助言を賜りました．本論文を井上教授のもとで完成させることができ，心より深く感謝しております．

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 松下誠 准教授には研究について様々なご意見を賜りました．いただいた御意見を参考に，論文を適切に修正することができました．松下准教授に深く感謝しております．

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 石尾隆 助教授には研究について様々なご意見を賜りました．石尾助教の御意見により研究内容の改善を行うことができ，深く感謝しております．

名古屋大学大学院情報科学研究科附属組込みシステム研究センター/情報システム学専攻 吉田則裕 准教授には，本研究の方針から論文の構成に至るまで直接の御指導及び御助言をして頂きました．本論文の完成は吉田准教授のおかげであると，深く感謝しております．

大阪大学大学院国際公共政策研究科 崔恩瀞 助教授には，研究及び研究生活について様々な御助言を賜りました．崔助教の御助言により，滞りなく研究を行うことができ，深く感謝しております．

井上研究室の先輩方におきましては，常に多くの御助言及び御指摘をして頂き深く感謝しております．特に大阪大学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 雑賀翼 氏，中村勇太 氏には，研究の方針や論文の書き方について様々な御助言を頂きました．

最後に，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研究室の皆様には，本論文の執筆にあたって様々な場面で支えて頂きました．皆様のおかげで本論文を完成させることができました．心より深く感謝しております．

参考文献

[1] The apache xerces project - xerces.apache.org. https://xerces.apache.org.

[2] Java platform, standard edition 8 api specification. https://docs.oracle.com/javase/8/docs/api/.

[3] Biegel B. and Diehl S. Highly configurable and extensible code clone detection. *in Proc. of the 17th Working Conference on Reverse Engineering (WCRE'10)*, 2010.

[4] Gorg C. and Weissgerber P. Detecting and visualizing refactorings from software archives. *in Proc. of the 13th International Workshop on Program Comprehension (IWPC'05)*, 2005.

[5] Martin F., Kent B., John B., William O., and Don R. *Refactoring:Improving the Design of Existing Code*. Addison Wesley, 1999.

[6] Shinpei H., Yasuyuki T., and Motoshi S. Search-based refactoring detection from source code revisions. *IEICE Trans. Inf. & Syst*, 2010.

[7] Levenshtein Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 1966.

[8] Prete K., Rachatasumrit N., Sudan N., and Kim M. Template-based reconstruction of complex refactorings. *in Proc. of the 26th International Conference on Software Maintenance (ICSM'10)*, 2010.

[9] Toshihiro K., Shinji K., and Katsuro I. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 2002.

[10] Kyle P., Napol R., Nikita S., and Miryung K. Catalogue of template refactoring rules. *in Proc. of the 26th International Conference on Software Maintenance (ICSM'10)*, 2010.

[11] Weissgerber P. and Diehl S. Identifying refactorings from source-code changes. *in Proc. of the 21th International Conference on Automated Software Engineering (ASE'06)*, 2006.

[12] Zeng W. and Church R. L. Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science 23*, 2009.

[13] 崔恩瀞, 藤原賢二, 吉田則裕, 林晋平. 変更履歴解析に基づくリファクタリング検出技術の調査. *J-STAGE*, 2015.

[14] 雑賀翼, 崔恩瀞, 後藤祥, 吉田則裕, 井上克郎. 組み合わせて実施されたリファクタリングの調査. 電子情報通信学会技術研究報告, Vol.114, No.23, pp. 1–6, 2014.

[15] 雑賀翼, 崔恩瀞, 吉田則裕, 春名修介, 井上克郎. Code smell の深刻度がリファクタリングに与える影響の調査. ソフトウェアエンジニアリングシンポジウム 2015 論文集, pp. 176–181, 2015.

付録

旧版コード

旧コミットのうち，本研究で探索が行われたDocumentImpl.javaとCoreDocumentImpl.javaのソースコードのうち，リファクタリングが行われた箇所との関連部分，本論文の例で使用された箇所の抜粋を載せる．

DocumentImpl.java

```
...省略
public class DocumentImpl
    extends CoreDocumentImpl
    implements DocumentTraversal, DocumentEvent, DocumentRange, DocumentLS {
...省略
    protected Vector ranges;
    protected Hashtable userData;
...省略
    protected void setUserData(NodeImpl n, Object data) {
        if (userData == null) {
            userData = new Hashtable();
        }
        if (data == null) {
            userData.remove(n);
        } else {
            userData.put(n, data);
        }
    }

    protected Object getUserData(NodeImpl n) {
        if (userData == null) {
            return null;
        }
        return userData.get(n);
    }
```

```
...省略
    public Range createRange() {
        if (ranges == null) {
            ranges = new Vector();
        }
        Range range = new RangeImpl(this);
        ranges.addElement(range);
        return range;
    }
}
```

CoreDocumentImpl.java

```
...省略
public class CoreDocumentImpl
    extends ParentNode implements Document {
...省略
    protected void setUserData(NodeImpl n, Object data) {
    }

    protected Object getUserData(NodeImpl n) {
        return null;
    }
...省略
}
```

新版コード

旧版コードと同様に，実験対象となったファイルのうちリファクタリングが行われた箇所の抜粋を載せる． **DocumentImpl.java**

```
...省略
```

```
public class DocumentImpl
    extends CoreDocumentImpl
    implements DocumentTraversal, DocumentEvent, DocumentRange, DocumentLS {
...省略
    protected Vector ranges;
...省略
    public Range createRange() {
        if (ranges == null) {
            ranges = new Vector();
        }
        Range range = new RangeImpl(this);
        ranges.addElement(range);
        return range;
    }
}
```

CoreDocumentImpl.java

```
...省略
public class CoreDocumentImpl
    extends ParentNode implements Document {
...省略
    protected Hashtable userData;
...省略
    public Object setUserData(Node n, String key,
                              Object data, UserDataHandler handler) {
        if (data == null) {
            if (userData != null) {
                Hashtable t = (Hashtable) userData.get(n);
                if (t != null) {
                    Object o = t.remove(key);
                    if (o != null) {
                        UserDataRecord r = (UserDataRecord) o;
```

```
                    return r.fData;
                }
            }
        }
        return null;
    }
    else {
        Hashtable t;
        if (userData == null) {
            userData = new Hashtable();
            t = new Hashtable();
            userData.put(n, t);
        }
        else {
            t = (Hashtable) userData.get(n);
            if (t == null) {
                t = new Hashtable();
                userData.put(n, t);
            }
        }
        Object o = t.put(key, new UserDataRecord(data, handler));
        if (o != null) {
            UserDataRecord r = (UserDataRecord) o;
            return r.fData;
        }
        return null;
    }
}

public Object getUserData(Node n, String key) {
    if (userData == null) {
        return null;
    }
    Hashtable t = (Hashtable) userData.get(n);
```

```
        if (t == null) {
            return null;
        }
        Object o = t.get(key);
        if (o != null) {
            UserDataRecord r = (UserDataRecord) o;
            return r.fData;
        }
        return null;
    }
...省略
    protected void setUserData(NodeImpl n, Object data) {
        setUserData(n, "XERCES1DOMUSERDATA", data, null);
    }

    protected Object getUserData(NodeImpl n) {
        return getUserData(n, "XERCES1DOMUSERDATA");
    }
...省略
}
```