特別研究報告

題目

Template Method パターン適用における類似メソッドの差分分離支援

指導教員
井上 克郎 教授

報告者
政井 智雄

平成 22 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 21 年度 特別研究報告

Template Method パターン適用における類似メソッドの差分分離支援

政井 智雄

内容梗概

ソフトウェアの保守を困難にしている要因の 1 つとして，コードクローンが指摘されている．コードクローンとは，ソースコード中に存在する同一，または類似したコード片のことであり，生成される原因の 1 つとしてコピーアンドペーストを用いた開発作業が挙げられる．例えばあるコード片に欠陥が含まれている場合，そのコードクローン全てに対し修正を検討する必要が生じる．近年，ソフトウェアが大規模になっており，このような作業には大きなコストがかかる．

コードクローンにかかる保守コストを低減させる方法の 1 つとして，ソフトウェアからコードクローンを集約し，取り除く方法が挙げられる．コードクローンにはいくつか分類が考えられているが，それらの 1 つに不一致部分（以降，ギャップ）を含むコードクローンがある．ギャップを含むコードクローンは，あるコード片をコピーアンドペーストした後，開発者が文の挿入や削除を行うことで作成される．ギャップを含まないコードクローンに比べると，ギャップを含むコードクローンの集約はギャップの修正，除去を考慮する必要があるため困難であり，修正作業の効率が低下する場合がある．

そこで本研究では，ギャップを含むコードクローンを集約するために，デザインパターンの 1 つである Template Method パターンの適用を支援する手法を提案する．Template Method パターンとは，親クラスで処理のフレームワークを定め，子クラスでその処理の具体的内容を定めるデザインパターンである．

本手法では Template Method パターンの適用を，ギャップを含み，メソッドとして抽出することが可能なコード片の候補を提示することで支援する．

オープンソースソフトウェアに存在するギャップを含むコードクローンに対し本手法を適用し，提示される候補を用いて Template Method パターンの適用を行った．適用の前後で，外部的動作に変化が無いことを確認できたため，本手法の有効性を確認できた．

主な用語

コードクローン

Tmplate Method パターン

リファクタリング
ソフトウェア保守

目次

1 まえがき **4**

2 背景 **7**

2.1 コードクローン 7

2.1.1 発生原因 7

2.1.2 定義 8

2.2 デザインパターン 9

2.3 リファクタリング 10

2.4 デザインパターンを利用するリファクタリング 11

2.5 抽象構文木 (Abstract Syntax Tree) 14

3 提案手法 **17**

3.1 [ステップ 1] 抽象構文木の生成 17

3.2 [ステップ 2] ギャップとなっている部分木の検出 17

3.3 [ステップ 3] 抽出可能な部分木列の検出 19

3.4 [ステップ 4] 範囲を拡大した複数の部分木列の検出 21

3.5 [ステップ 5] 部分木列の分類 22

3.6 [ステップ 6] コード片の提示 23

4 適用実験 **24**

4.1 準備 24

4.2 提案手法を用いたリファクタリング 25

4.3 考察 26

5 関連研究 **28**

6 まとめと今後の課題 **29**

謝辞 **30**

参考文献 **31**

付録 **33**

A 類似文字列マッチングアルゴリズム **33**

1 まえがき

ソフトウェアの保守を困難にしている要因の 1 つとして，コードクローンが指摘されている．コードクローンとは，ソースコード中に存在する同一，または類似したコード片のことであり，開発，保守の工程において，あるコード片に修正を行う必要が生じた場合，そのコードクローン全てに対して修正を検討しなくてはならない．しかし近年のソフトウェアの大規模化，複雑化に伴ない，このような作業にかかるコストは非常に大きくなっている．したがってコードクローンを効率的に集約し，取り除くことができれば，これらのコストを低減させることが可能である．

コードクローンは Bellon の定義に基づくと，3 つのタイプに分類することができる [3].

タイプ 1 空白やタブの有無，括弧の位置などのコーディングスタイルを除いて，完全に一致するコードクローン．

タイプ 2 関数名や変数名などのユーザ定義名，変数の型などの一部の予約語のみが異なるコードクローン．

タイプ 3 タイプ 2 における変更に加えて，文の挿入や削除，変更が行われたことで，不一致部分を（以降，ギャップ）を含むコードクローン．

タイプ 3 のコードクローンは，ギャップが存在するために単純に 1 つに集約することができない．したがってタイプ 3 のコードクローンは，取り除くことがタイプ 1，タイプ 2 のコードクローンに比べて難しい．

ソフトウェアからコードクローンを取り除き，保守性を高めるための手段の 1 つとして，リファクタリングが挙げられる [13]．リファクタリングとは“ 外部から見たときの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 ”[6] である．メソッドの抽出や，フィールドの移動などの繰り返し行われているとされるリファクタリングは，パターンとして書籍 [6] やウェブサイト [5] にまとめられている．このようなパターンはリファクタリングパターンと呼ばれ，適用を検討すべきソースコードの特徴や適用手順などがまとめられている．例えばメソッドの抽出は，あるメソッド内の特定の動作を行うコード片を同一クラス内に別メソッドとして抽出することで，周辺のソースコードとの関係を理解しやすい構造に変化させるリファクタリングパターンである．

リファクタリングに関して，Kerievsky はデザインパターンを利用する手法を提案している [11]．ソフトウェア開発におけるデザインパターンとは，ソフトウェア開発時に生じる典型的な問題に対する過去の設計者が編み出し蓄積した解決策で，再利用しやすいように名前をつけたものをいう．特にデザインパターンとしては，Gamma らが定義している GoF デザインパターン [7] が広く知られている．Kerievsky が文献 [11] の中で提案している 27 種類の

リファクタリングパターンのうち，7 つのリファクタリングパターンがコードクローン（文献 [11] 内では重複コード）に対し効果的である分類されており，その中の 1 つに“ Template Method の形成 ”がある．

“ Template Method の形成 ”とは，GoF デザインパターンの 1 つである Template Method パターンに基づくリファクタリングパターンである [6, 11]．Template Method パターンとは，親クラスで処理のフレームワークを定め，子クラスでその処理の具体的内容を定めるデザインパターンである．親クラスに処理の順序を決定するメソッドが定義され，そのメソッドの記述には，同じクラス内で定義された抽象メソッドが用いられており，この抽象メソッドの処理の内容は子クラスによって実装される．それぞれの子クラスの実装には共通した処理を実装する必要がなく，子クラス間で類似したコード片が現れることを回避している．“ Template Method の形成 ”では，処理のステップが類似した 2 つのメソッドについて，まず異なる処理のステップをそれぞれ親クラスで定義された抽象メソッドの実装として抽出することで記述を揃える．次に記述を揃えたメソッドを共通の親クラスへ引き上げて共通な処理の順序を実装するメソッドを形成する．このリファクタリングパターンにおける利点の 1 つは，異なる処理を含む類似メソッド，つまりタイプ 3 のコードクローンを含む 2 つのメソッドに対しても，そのコードクローンを取り除くことができる点である．

この利点に注目し，本研究では，この“ Template Method の形成 ”を用いて，そのリファクタリングを支援する手法を提案する．類似した 2 つのメソッドに対する“ Template Method の形成 ”は以下の 4 つのステップに分けられる．

ステップ 1 メソッド間のギャップを検出する．

ステップ 2 抽象メソッドの実装として，ギャップを包括するような抽出するコード片を決定する

ステップ 3 ステップ 2 で決定したコード片を抽出する．

ステップ 4 記述を揃えたメソッドを親クラスに引き上げる．

本手法では，これらのステップのうち，以下のようにステップ 1 とステップ 2 を支援する．

ステップ 1 の支援 ソースコードから生成した抽象構文木を用いて 2 つのメソッドの比較を行い，抽象構文木の部分木の形でメソッド間のギャップを自動的に検出する．

ステップ 2 の支援 検出したギャップを用いて，ギャップを含みかつメソッドとして抽出が可能なコード片を探索し，抽象メソッドの実装として抽出するコード片の候補を複数提示する．

複数の候補を提示する理由は，抽出できるコード片の候補の中から最良の候補を選出する明確な基準が無いため，できるだけ多くの候補を探索し提示することで，開発者の判断を交えたより良い決定を促すことができると考えられるからである．この提案手法は統合開発環境 Eclipse[4] に組み込んで利用する Eclipse プラグイン（Eclipse に統合できる新しい機能の実装）として実装し，Eclipse を用いた開発中の必要な時に利用できるようにした．この Eclipse プラグインを用いて，オープンソースソフトウェアの中から選出した 2 つの類似したメソッドを対象とした適用実験を行った．

提示されたコード片の候補を用いて，実際に“ Template Method の形成 ”を行い，そのリファクタリングの前後でプログラム全体の外部的動作に変化が無いことを確認できたため，提示された候補の有効性を確認できた．

以降，2 節では本研究に関わる用語であるコードクローン，デザインパターン，リファクタリング，及び抽象構文木について説明する．3 節では，リファクタリングを支援する本研究の提案手法について詳細に説明し，4 節では本手法を用いた適用実験の結果を，5 節では本研究についての考察を，6 節で関連研究を述べ，最後に 7 節で本研究のまとめと今後の課題について述べる．

2 背景

本研究の提案手法の背景として，本研究の問題となるコードクローン，及び本手法に用いるデザインパターン，リファクタリング，デザインパターンを利用するリファクタリング，抽象構文木について説明する．

2.1 コードクローン

コードクローンとは，ソースコードの中に存在する同一，または類似したコード片のことである．コードクローンが存在するプログラムでは，欠陥が見つかったコード片を修正する場合，そのコード片のコードクローン全てに対して修正を検討しなければならない．しかしこのような作業は，近年のソフトウェアの大規模化，複雑化に伴ってかかるコストが大きくなっている．そのため，コードクローンを効率的に把握する，または取り除く技術が必要とされている [8].

2.1.1 発生原因

コードクローンが発生する原因は以下の項目が挙げられる [2, 12].

既存コードのコピーアンドペーストによる再利用 正常な動作が確認されている既存のソースコードを流用し，部分的に変更を加える方が，一からコーディングを行うよりも信頼性が高い．したがってコピーアンドペーストによる既存コードの再利用が多くなっている．

定型処理 キューの挿入処理やデータ構造へのアクセス処理など，定義上簡単で頻繁に用いられる処理はコードクローンになる傾向がある．

プログラミング言語の仕様 抽象データ型や，ローカル変数を用いることができない場合には，同じようなアルゴリズムを持つ処理を繰り返し書かなくてはならない場合がある．

パフォーマンスの改善 リアルタイムシステムなど時間制約のあるシステムにおいて，インライン展開などの機能が提供されていない場合に，特定のコード片を意図的に繰り返し記述することで，パフォーマンスの改善を図ることがある．

コード生成ツールの生成コード コード生成ツールは，あらかじめ定められたコードをベースにして自動的にコードを生成する．そのため目的の処理が類似している場合，識別子等を除いて類似したコードが生成される．

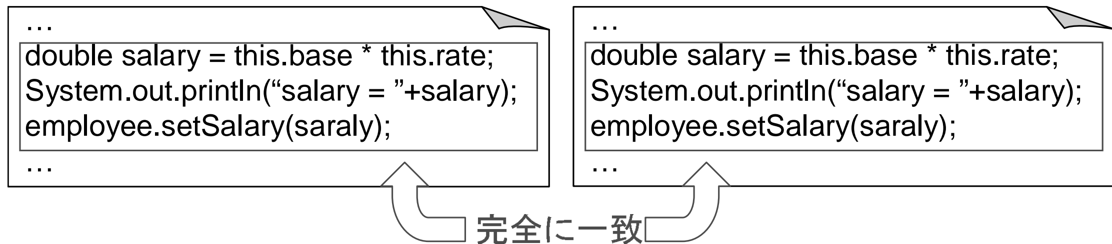

図 1: タイプ 1 のコードクローンの例

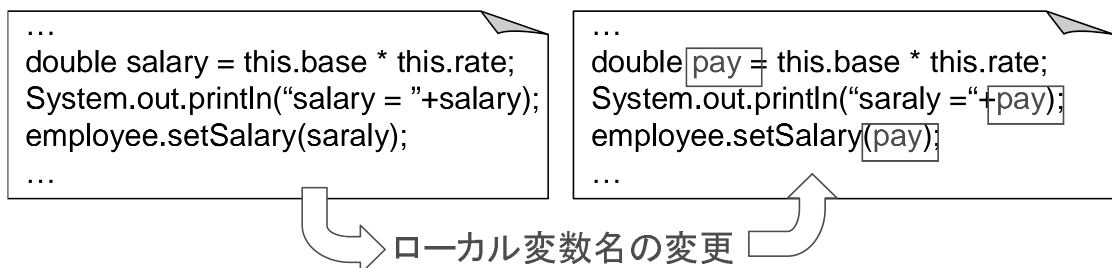

図 2: タイプ 2 のコードクローンの例

複数のプラットフォームに対応したコード 複数の OS（Windows，Macintosh，Linux，FreeBSD など）や CPU（i386 系，amd64 系など）に対応したソフトウェアは，各プラットホーム用のコード部分に重複した処理が存在する傾向がある．

2.1.2 定義

コードクローンには様々な定義がある．というのも，これまでに様々なコードクローン検出手法が提案されており，それらはどれも異なったコードクローンの定義をもつからである．つまり，コードクローンの厳密で普遍的な定義は存在しない．Bellon は，コードクローン間の相違の度合いに基づいた 3 つの分類を定義している [3].

タイプ 1 空白やタブの有無，括弧の位置などのコーディングスタイルを除いて，完全に一致するコードクローン（図 1，参照）.

タイプ 2 変数名や関数名などのユーザ定義名，また変数の型などの一部の予約語のみが異なるコードクローン（図 2，参照）.

タイプ 3 タイプ 2 における変更に加えて，文の挿入や削除，変更が行なわれたコードクロー

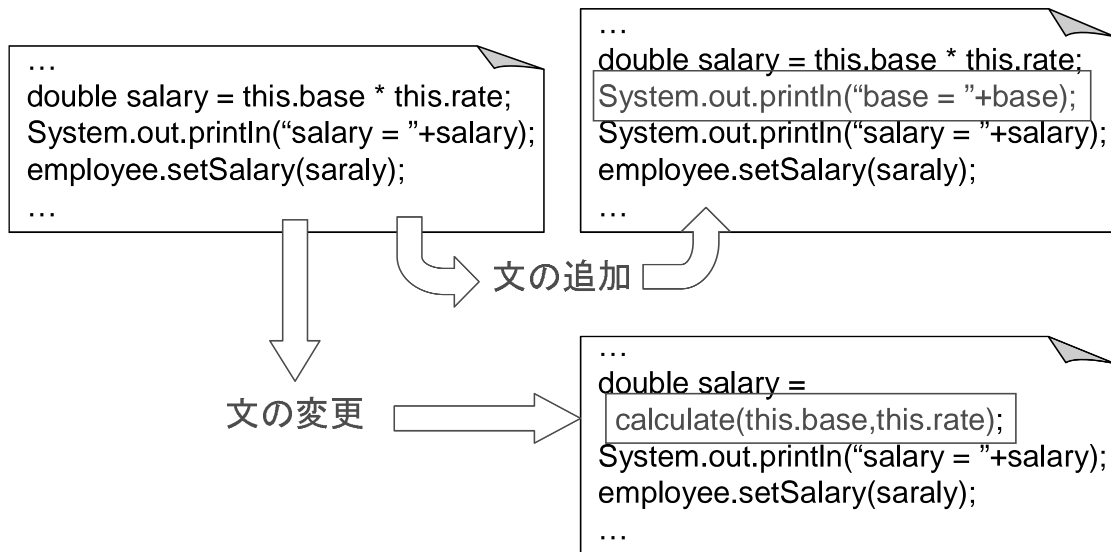

図 3: タイプ 3 のコードクローンの例

ン（図 3，参照）．このような不一致部分を含むコードクローンを以降，ギャップを含むコードクローンと呼ぶ.

2.2　デザインパターン

デザインパターンとは, “ 種々の状況における設計上の一般的な問題の解決に適用できるよう，オブジェクトやクラス間の通信を記述したもの ”と定義されている [7]．一般に，パターンは次の 4 つの基本的な要素を有している [7].

パターン名 設計問題とその解法及びその結果を 1,2 語で記述した名称．パターンを習得した技術者同士ならばパターン名を使うことで意思の疎通が容易になり，議論や，文書への記録時に役立つ.

問題 どのような場合にパターンを適用すべきかを記述したもの.

解法 設計問題を抽象的に記述し，クラスやオブジェクトなどの要素の配置によってどのようにそれらの問題を解決するかを示している.

結果 パターンを適用する際の結果やトレードオフを記述する.

一般に有名なデザインパターンに GoF デザインパターンがある．GoF デザインパターンの例として，本研究で利用するデザインパターンである Template Method パターンの概要を説明する.

Template Method パターン

Template Method パターンとは，共通な処理の流れを親クラスで実装し，処理の具体的な実装は子クラスで行うデザインパターンである．親クラスのメソッドで処理の順序を実装するが，そのメソッド内で呼び出されている一部のメソッドは同じクラス内で抽象メソッドとして定義しており，具体的な動作は実装していない．これらの抽象クラスをそれぞれの子クラスでオーバーライドすることにより，目的に応じた子クラスごとに異なる処理を実装することができる．共通な処理の流れを親クラスで実装することで，共通な処理を重複して記述せずに子クラスを実装できる．さらに子クラスにおける，親クラスのメソッドに対する拡張を制限することができ，欠陥の混入を抑えることができる．また同じ処理の流れを利用する新たな子クラスを追加する際には，同様に抽象メソッドをオーバーライドするだけでよく，拡張も容易となっている．

2.3 リファクタリング

リファクタリングとは“ 外部から見たときの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 ”[6] である．Fowler は設計の問題を匂いに例え，コードの不吉な匂い（Bad Smell）と呼んでおり，その不吉な匂いを解消する為にどのようなリファクタリングが最も役に立つかを説明している．コードの不吉な匂いとされる設計の問題の一つにコードクローン（文献 [11] ではコードの重複）があり，その対処として以下のようなリファクタリングパターンが挙げられている．

メソッドの抽出 (Extract Method)

メソッド内のコード片を抽出し，新しいメソッドとして定義するリファクタリングパターンである．抽出されたコード片は，新しく定義したメソッドの呼び出し文に置き換える．同じクラス内に，抽出したコード片のコードクローンがある場合は，同様にメソッドの呼び出し文に置き換えることでコードクローンを取り除くことができる（図 4 参照）．メソッド内から抽出するコード片は，変数の代入や参照に応じて引数として受け渡す変数，戻り値とする変数を明確に決める必要があり，これらが明確に定まらない場合，抽出することはできない．

メソッドの引き上げ (Pull Up Method)

子クラスで定義されたメソッドを親クラスへと引き上げるリファクタリングパターンである．同じ親クラスを持つ兄弟クラスに同じ処理を行うメソッドが定義されている場合，それらを親クラスへと引き上げることで，コードクローンを取り除くことができる．メソッド内

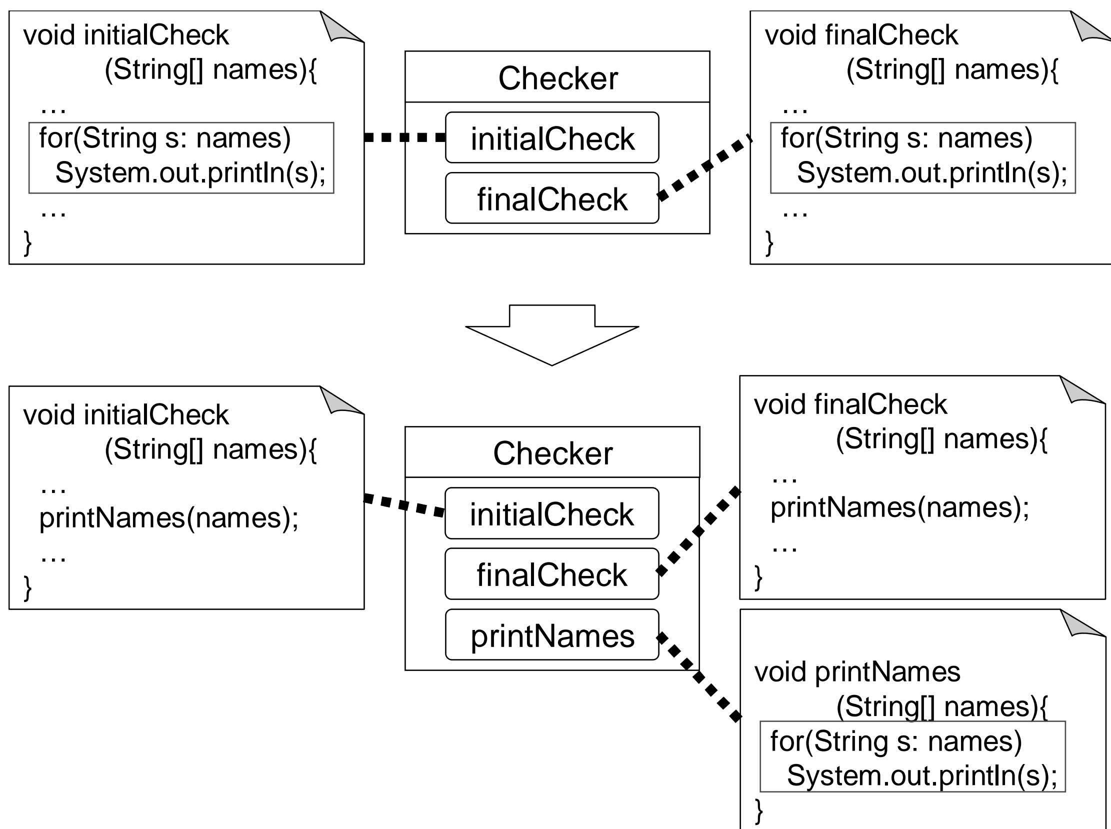

図 4: コードクローンを対象としたメソッドの抽出

の一部がコードクローンである場合は，上記のメソッドの抽出と組み合わせ，抽出したメソッドを引き上げることでコードクローンを取り除くこともできる．（図 5 参照）

2.4　デザインパターンを利用するリファクタリング

Kerievsky は文献 [11] の中で，ソフトウェアの設計の段階において，デザインパターンの構造を目指したコーディングは，あくまでデザインパターンの構造を作るだけであり，設計者の抱えているニーズに適しているとは言えない．必要に応じて，もっとも状況に適したデザインパターンを選び，実装に取り入れるよう設計をリファクタリングすることが望ましい，と述べている．以下に，コードクローンを取り除く為のデザインパターンを利用するリファクタリングパターンとして，“ Template Method の形成 ”を説明する．

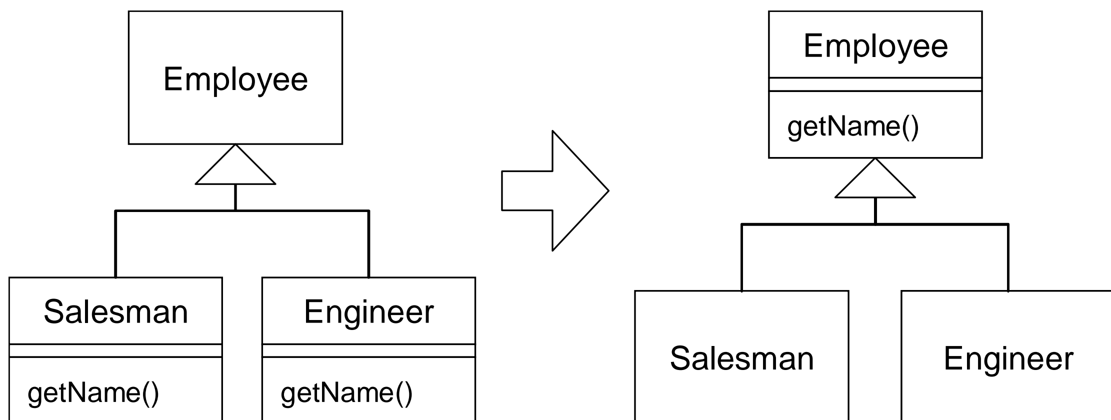

図 5: コードクローンを対象としたメソッドの引き上げ [6]

Template Method の形成 (Form Template Method)

“ Template Method の形成 ”は，親クラスが共通で，類似した処理の流れのメソッドをもつ子クラス間に，Template Method パターンを適用するリファクタリングパターンである．“ Template Method の形成 ”は，以下の 4 つのステップに分けられる．

ステップ 1 メソッド間のギャップを検出する．

ステップ 2 抽象メソッドの実装として，ステップ 1 で検出したギャップを包括するように抽出するコード片を決定する．

ステップ 3 ステップ 2 で決定したコード片を抽出する．

ステップ 4 記述を揃えたメソッドを親クラスに引き上げる．

これらのステップを，図 6 を用いて説明する．

ステップ 1　図 6 の類似した 2 つのメソッド (ResidentialSite クラスと LifelineSite クラスの getBillableAmount メソッド) では，戻り値とする式は同じだが，その式に用いられる 2 つの変数のローカル宣言文に違いがあるため，これらをギャップとする．

ステップ 2　ステップ 1 で検出したギャップを吸収できるように，抽出するコード片を決定する．図 6 においては，変数 base の宣言文及び参照，変数 tax の宣言文及び参照をそれぞれ抽出するコード片とする．

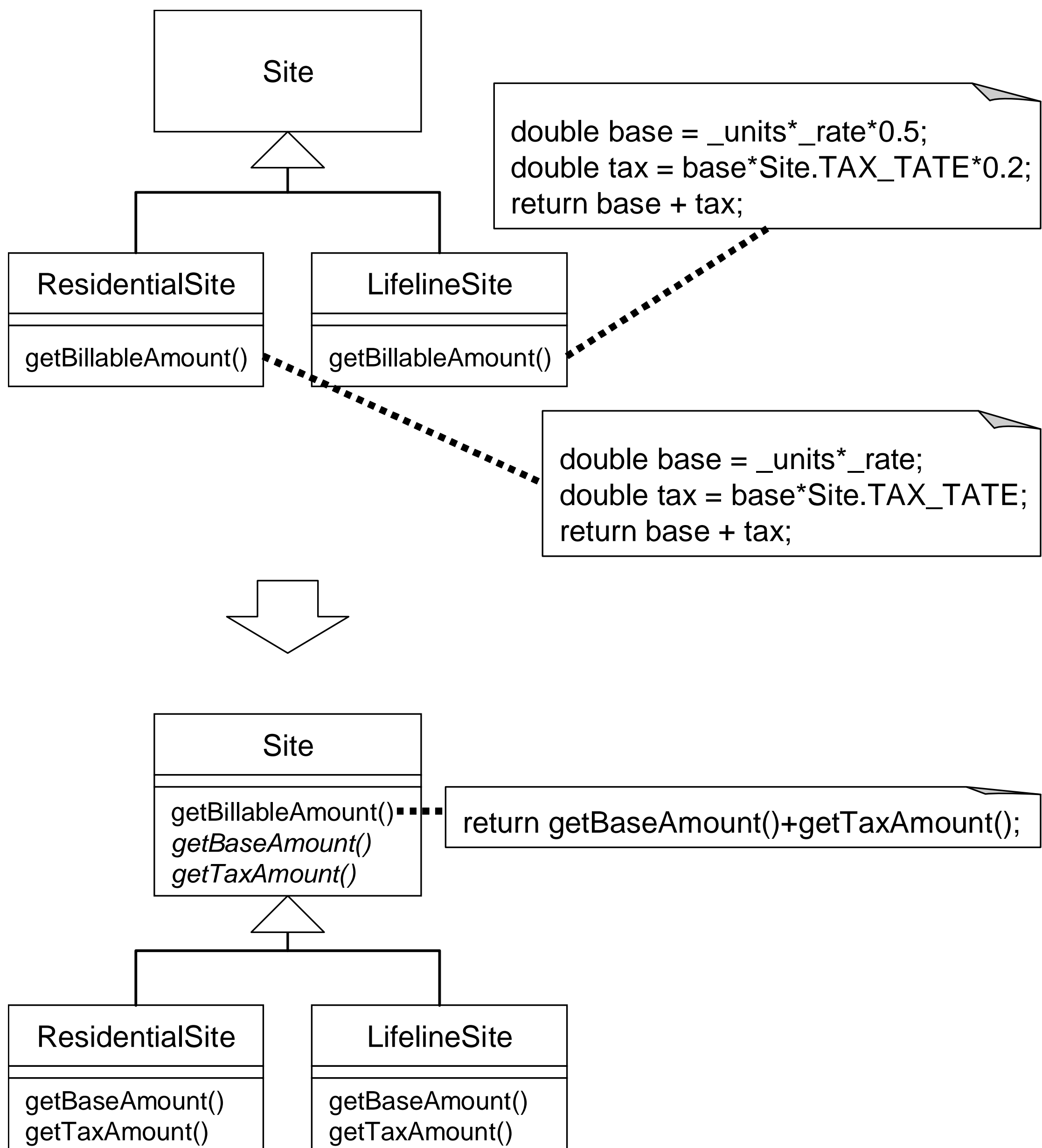

図 6: Template Method の形成 [6]

ステップ 3　ステップ 2 で決定した抽出するコード片を，実際にメソッドとして抽出する．この際，抽出したコード片で定義されるメソッドのシグニチャ，呼び出し時に引数として渡す変数，戻り値の型，及び値を戻す変数は同じでなくてはならない．シグニチャとは，呼び出し時にメソッドを特定するために必要な，メソッド名と引数の型の列である．同じシグニチャ，戻り値の型の抽象メソッドを親クラスで定義しておくことで，各子クラスでのメソッ

ドの実装は抽象メソッドをオーバーライドする形になる．図 6 においては，まずそれぞれの変数の宣言文を抽出し，2 つのメソッド（getBaseAmount メソッド，getTaxAmount メソッド）を定義し，同じシグニチャ，戻り値の型の抽象メソッドを親クラス（Site クラス）に定義する．次に，変数の宣言文を消去し，変数の参照をそれぞれのメソッド呼び出し文に置き換えることで記述を揃えている．

ステップ 4　ステップ 3 で記述を揃えたメソッド（getBillableAmount）を，親クラスへと引き上げることで，Template Method パターンが適用されたソースコードに修正を終えた．上記の 4 つのステップで“ Template Method の形成 ”がなされ，コードクローンを取り除くことができた．このリファクタリングにより，図 6 の Site メソッドの子クラスを新たに追加する場合，リファクタリング前ではコードクローンを含むメソッドが増えてしまうが，リファクタリング後は 2 つの抽象メソッドを実装するだけで子クラスを追加できるため，拡張性の高い設計となっている．

2.5　抽象構文木 (Abstract Syntax Tree)

抽象構文木とは，プログラムを“ 文（Statement）”や“ 式（Expression）”,“ 識別子（Identifier）” などの分類用語を用いた，コンパイラごとに異なる抽象構文（Abstract Syntax）によって生成される構文木である．節点に演算子，葉にそのオペランドを対応させた木構造であり，葉は変数や定数に対応する．統合開発環境 Eclipse[4] には，ソースコードから抽象構文木を生成する機能があり，図 7 はその機能を用いて生成した抽象構文木の例である．この抽象構文木は，ASTNode クラスを継承した以下のようなクラスのインスタンスを節点（以降，AST ノード）として生成される．

MethodDeclaration メソッドの宣言に対応する AST ノード．修飾子，戻り値の型，型パラメータ，メソッド名，引数宣言，処理にそれぞれ対応する AST ノードを子に持つ．

Modifier 修飾子（public，final など）に対応する AST ノードをノード．修飾子を表す String 型の値を記憶しており，子ノードを持たない．

SimpleName メソッド名や変数名など，ユーザ定義名に対応する AST ノード．名称を表す String 型の値を記憶しており，子ノードを持たない．

PrimitiveType 基本データ型（int，char など）の記述に対応する AST ノード．型名を表す String 型の値を記憶しており，子ノードを持たない．

SingleVariableDeclaration 引数変数の宣言など，複数の変数をまとめて宣言することがで

きず，また初期値を設定できない変数宣言に対応する AST ノード．変数の型と変数名にそれぞれ対応する AST ノードを子に持つ．

Block メソッド宣言や，if 文などに用いられる“｛”と“｝”に囲まれたステートメントの列に対応する AST ノード．列を成す各ステートメントに対応する AST ノードを列として子に持つ．

ExpressionStatement 変数宣言文や if 文などの子として存在できる式である代入文やメソッド呼び出し文，変数のインクリメント文などが，単一の文して記述されている場合に対応する AST ノード．代入やメソッド呼び出しのような式に対応する AST ノードを子に持つ．

Assignment 代入式に対応する AST ノード．代入される左辺の変数，代入する式にそれぞれ対応する AST ノードを子に持ち，代入演算子を表すクラスのインスタンスを値に持つ．

InfixExpression $x * y$ のような中置構文式に対応する AST ノード．被演算子に対応する AST ノードを子に持ち，演算子を表すクラスインスタンスを値を記憶している．さらに拡張として，同じ演算子での計算が続いて記述されている場合は，演算子に対応する AST ノードを複数子に持つこともできる．

ReturnStatement return 文に対応する AST ノード．戻り値とする式に対応する AST ノードを子に持つ．

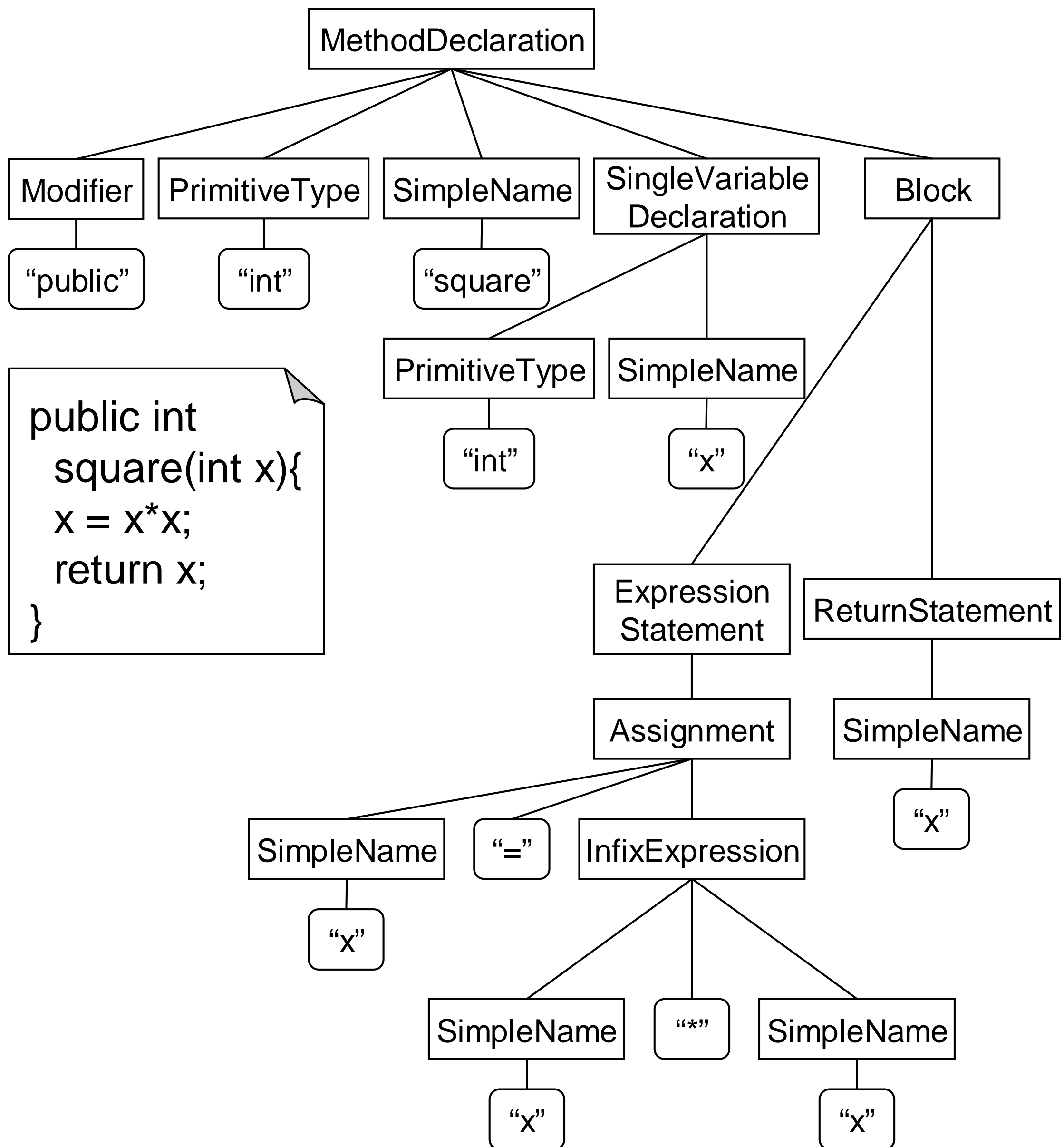

図 7: Eclipse が生成する抽象構文木

3　提案手法

本節では，提案手法の概要，また提案手法における実装を述べる．

本手法は入力としてギャップを含むコードクローンを互いに含む 2 つの類似メソッドが与えられ，動作は以下の 6 つのステップに分けられる．図 8 は提案手法全体の流れを示す.

ステップ 1 2 つのメソッドのソースコードから，抽象構文木を生成する．

ステップ 2 2 つの抽象構文木間のギャップとなっている部分木を検出する．

ステップ 3 ギャップとなっている部分木を含み，メソッドとして抽出可能な部分木列を検出する．

ステップ 4 ステップ 3 で検出した部分木列から範囲を拡大した，可能な限り多くの抽出可能な部分木列を検出する．

ステップ 5 ステップ 3，ステップ 4 で検出されたメソッドとして抽出可能な部分木列（のセット）を，実際に抽出を行った後，記述を揃えることが容易か否かにより分類する．

ステップ 6 ステップ 5 で行った分類に従い，メソッドとして抽出可能な部分木（のセット）に対応するソースコード中のコード片を提示する．

3.1　[ステップ 1] 抽象構文木の生成

Eclipse の抽象構文木生成の機能を用いて，用意された 2 つのメソッドの抽象構文木を生成する（図 7，参照）．

3.2　[ステップ 2] ギャップとなっている部分木の検出

ステップ 1 で生成された 2 つの抽象構文木の比較を行い，抽象構文木間のギャップとなっている部分木を検出する．

抽象構文木の比較は，主に以下の 3 つの操作からなる．

AST ノードの種類の比較 2 つの AST ノードの種類が同じかを比較する．異なる場合は比較した 2 つの AST ノードを対応するギャップとし，種類が同じ場合は値，及び子ノードの比較を行う．

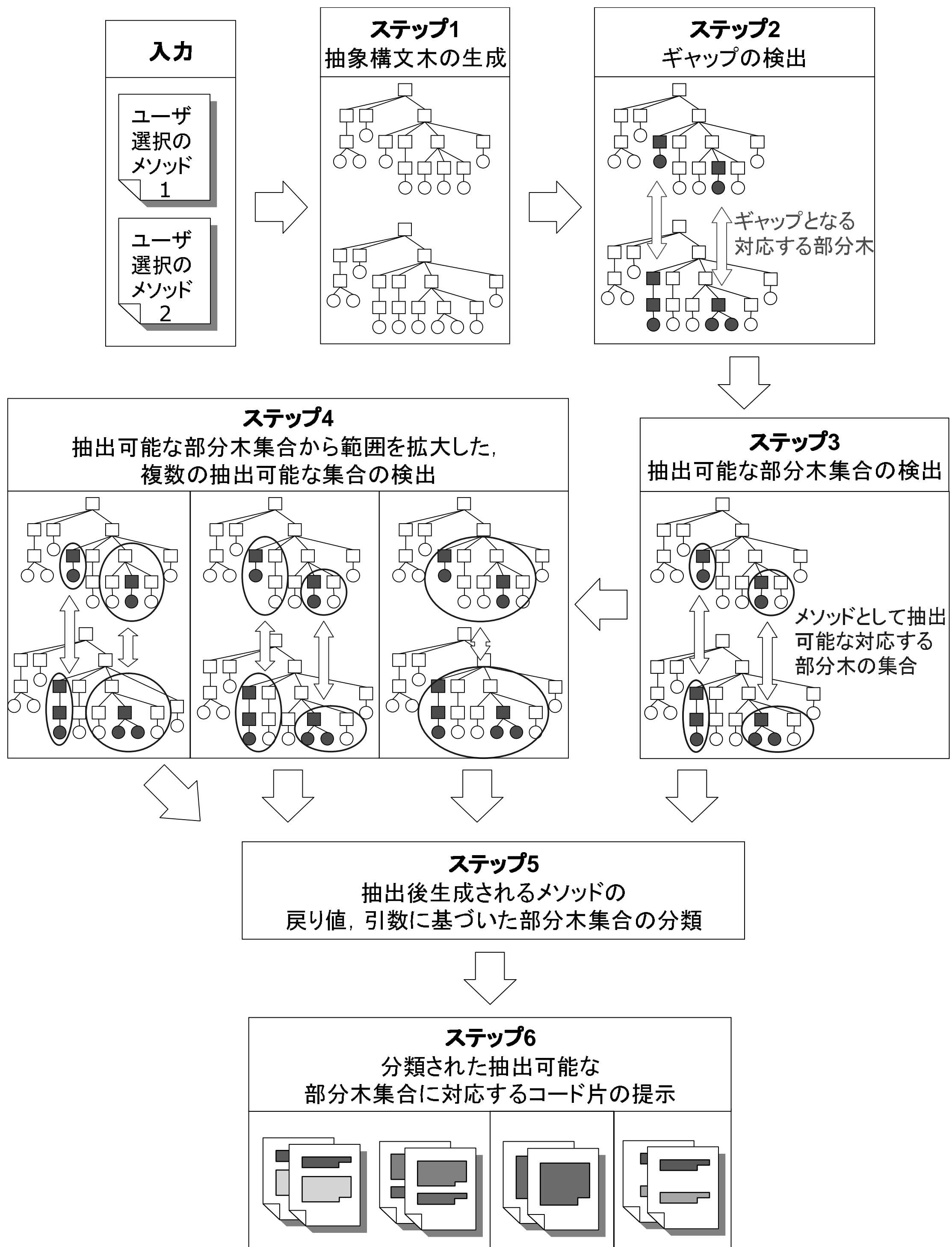

図 8: 提案手法全体の流れ

ASTノードの値，子ノードの比較 ASTノードの持つ値，及び子ノードを比較する．値が異なる場合は比較した2つのASTノードを対応するギャップとする．子ノードを持つ場合は，子ノードに対し順にASTノードの種類の比較から同様に行う．子ノード同士がギャップであった場合は，親ノードの種類によって，ギャップとする場合（代入文の左辺など）と，ギャップとしない場合（if文の条件式，代入式の右辺など）がある．ただし節2.5で説明したBlockノードを比較する場合は，下記のASTノード列の比較を行う．

ASTノード列の比較 節2.5で説明したBlockノードが持つ子ノードの列を比較する場合に行う．上記2つのASTノードの比較と既存の類似文字列マッチングアルゴリズムを用いて，ASTノード列の中から対応するギャップとなるASTノードを検出する．Blockノードは，子ノード全てがギャップとなっている場合のみBlockノードを含む部分木全体をギャップと判断する．

上記の3つの操作を，抽象構文木の根である，メソッド宣言に対応するASTノード（節2.5で説明したMethodDeclaration）から行うことで，抽象構文木全体に対し比較を行い，ギャップとなるASTノードを検出する．ギャップと判断されたASTノードの子ノードはすべてギャップと判断するため，ギャップとなるASTノードを以降，ギャップとなる部分木と呼ぶ．

ASTノード列の比較に既存の類似文字列マッチングアルゴリズムを用いた理由は，ASTノードの列を比較する際，単純に先頭から順にノード同士の比較を行うのでは，同一のノードが多い場合であっても，同一のノードが互いに列における位置が異なるだけで，比較も行われずに差分と判定されてしまい，ギャップである部分木の数が非常に多くなる可能性があるからである．

既存の類似文字列マッチングアルゴリズムは，与えられた2つの文字列がどれだけ類似しているか，を計算するアルゴリズムである．片方の文字列に，1文字を削除，1文字を挿入，1文字を置換，という3つの操作を最低何度行えばもう片方の文字列と同一の文字列へ変化させられるかを調べることで，2つの文字列の類似度を測ることができる．またこれらの必要な操作を調べることで，文字列内の共通部分文字列，及びギャップである文字を把握することができる．ASTノード列の比較において，このアルゴリズムを用いることで，同一であるノード，ギャップであるノードを，全て比較を行った上で把握することができ，ギャップであるASTノードの数を，単純な列の比較に比べ少なくすることができる．

3.3 [ステップ3] 抽出可能な部分木列の検出

ギャップである部分木を含み，メソッドとして抽出可能な部分木列を検出する．部分木列に対応するソースコード中のコード片が，メソッドとして抽出可能か否かはEclipseのメソッ

ド抽出機能を用いて判定する．Eclipse のメソッド抽出機能が，メソッドとして抽出することが不可能であると判定する条件の中で，本手法に関係するものは以下の 3 つである．

条件 1 複数の変数の宣言文，または変数への代入文が含まれており，それらの変数が抽出するコード片の後で参照されている．
メソッドとして抽出する際に戻り値が複数必要になるが，戻り値を複数としたメソッドは生成できないため抽出することができない．

条件 2 抽出するコード片に break 文，continue 文が含まれているが，それらに対応する制御文が含まれていない．
break 文，continue 文のみをメソッドとして抽出しても，処理を変化させる制御文が存在せず，コンパイルエラーが発生する，または動作が変化するため抽出することができない．

条件 3 構文木を生成しているメソッドの戻り値が void 型であり，かつ抽出するコード片が return 文を含んでいる．
抽出すること自体は可能であるが，抽出した部分と置き換える記述に return 文を記述することができず，プログラムの元の動作が変化してしまう可能性があるため抽出できないものとする．

ギャップとなる部分木に対応するソースコード中のコード片が，メソッドとして抽出可能であった場合は，次のギャップとなる部分木にも同様に判定を行う．メソッドとして抽出が不可能と判定された場合には，以下の操作の，操作 1 と操作 2 を同時を開始し，メソッドとして抽出可能な部分木列を検出する（図 9 参照）．

操作 1 左（図 7 のような抽象構文木における左，ソースコード上では左，または上にある文）に隣り合う部分木を集合に含め，メソッドとして抽出可能かの判定を行う．部分木列が抽出可能であると判定されるまで，繰り返し同様の操作を行う．対応する部分木列の両方が抽出可能であると判定された場合は，その集合をメソッドとして抽出可能な部分木列として検出し，この操作を終了する．左に隣り合う部分木が存在しなかった場合，親ノードがメソッド宣言のブロックに対応するノードでなければ操作 3 を行う．

操作 2 右（ソースコード上では右，または下にある文）に隣り合う部分木を集合に含め，メソッドとして抽出可能かの判定を行う．部分木列が抽出不可能であると判定された場合，現在の部分木列に対して，操作 1 と操作 2 を行う．対応する部分木列の両方が抽出可能であると判定された場合は，その集合をメソッドとして抽出可能な部分木列として検出し，この操作を終了する．右に隣り合う部分木が存在しなかった場合，操作を終了する．

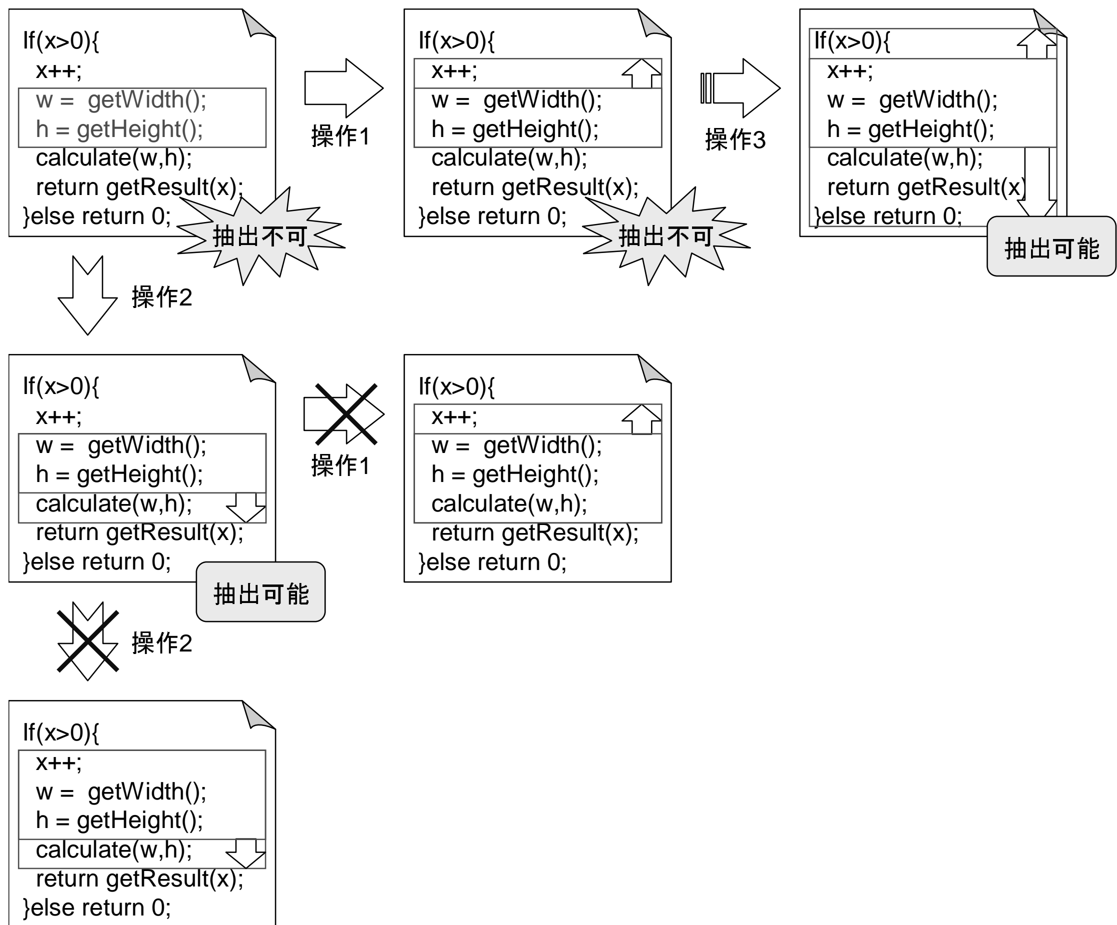

図 9: メソッドとして抽出可能な部分木列検出のための操作

操作 3 部分木列を空にし，親ノードを部分木列に含め，メソッドとして抽出可能かを判定する．抽出が不可能である判定された場合，この部分木列に対し操作 1 と操作 2 を行う．

上記の操作を全てのギャップとなる部分木に対して行い，それぞれに対してメソッドとして抽出可能な部分木列を検出する（図 9，参照）．

3.4 [ステップ 4] 範囲を拡大した複数の部分木列の検出

ステップ 3 で検出した部分木列の中から，選択された部分木列から範囲を拡大した，メソッドとして抽出可能な部分木列を可能な限り複数検出する．検出は，ステップ 3 と同様の操作で行う．ただし，メソッドとして抽出可能な部分木列が検出されても操作は終了しない．

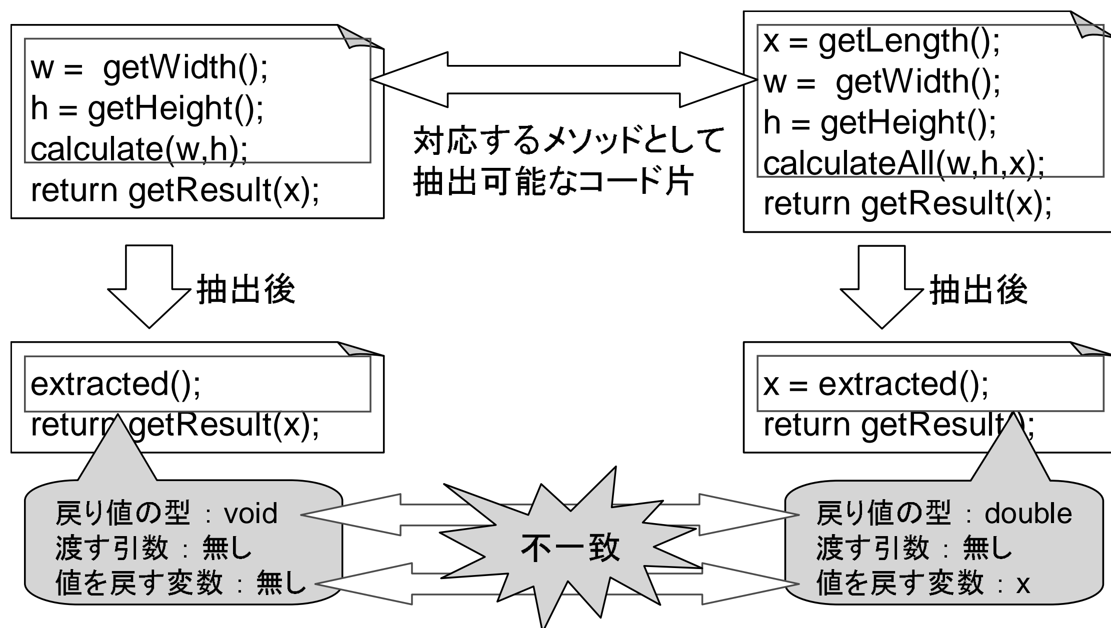

図 10: 抽出後の記述が一致しない例

3.5 [ステップ 5] 部分木列の分類

検出されたメソッドとして抽出可能な部分木に対応するコード片を抽出した場合に生成されるメソッドの戻り値，引数として渡す変数，及び値を戻す変数を調べる．これらが，対応する抽出箇所で互いに異なる場合，単純なメソッドの抽出で記述を揃えることができず，容易に Template Method を形成することができない（図 10，参照）．以下の条件を用いて，メソッドとして抽出可能な部分木に対応するコード片を 5 つに分類する．

条件 1 戻り値の型，または値を戻す変数が異なる．

条件 2 引数として渡す変数が異なる．

条件 3 片方にのみ対応する位置にコード片がない

分類 1 条件 1，条件 2 を満たす抽出箇所が存在しない．

分類 2 条件 1 を満たすが抽出箇所が 1 つ存在する．

分類 3 条件 2 を満たす抽出箇所が 1 つ存在する．

分類 4 条件 1，及び条件 2 を満たす抽出箇所が 1 つ存在する．

分類 5 条件 3 を満たす抽出箇所が１つ存在する．

分類 6 条件 1〜3 のうちいずれか，もしくは複数を満たす抽出箇所が複数存在する．

これらの分類のうち，分類 1 はそのままメソッドの抽出を行うことで記述を揃えることができ，Template Method の形成が容易といえる．分類 2〜5 は，抽出を行うだけでは，1 箇所記述が揃わず，分類 6 に至っては複数箇所の記述が揃わないため，メソッドの抽出を行う前にソースコードを修正する必要があり，Template Method の形成が容易とはいえない．したがって，これらを分類することで，Template Method の形成が容易な候補の理解を促すことが可能となる．

3.6 [ステップ 6] コード片の提示

ステップ 5 で行った分類に従い，メソッドとして抽出可能な部分木に対応するソースコード中のコード片を提示する．コード片は，Eclipse におけるウィザードで表示することで提示する．

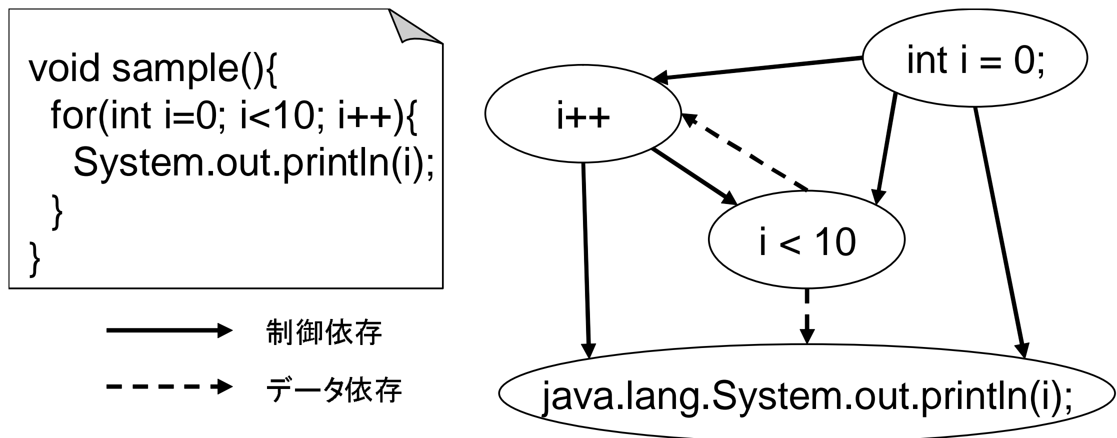

図 11: ソースコードから生成されるプログラム依存グラフの例 [9]

4　適用実験

本節では，3 節で説明した提案手法を，実際のソースコードに適用した事例について述べる．

提案手法は，リファクタリングを支援することを目的としている．そのため，提示される候補がリファクタリングに有効であり，また必要な作業に応じて候補が正しく分類されている必要がある．したがって本実験の目的は，提示される候補の有効性，および候補に対して行う分類の有効性をそれぞれ確認することとした．

まず適用対象及びその準備について説明する．続いて，実際に提案手法により分類して提示された候補を用いてリファクタリングを行った結果と，その考察を述べる．

4.1　準備

適用実験には，Java，C#，C++等に対応したコンパイラ・コンパイラであるオープンソースソフトウェアの ANTLR2.7.4[1] を対象に用いた．また，ANTLR2.7.4 から，提案手法を適用する類似メソッドを検出するためのコードクローン検出ツールとして，Scorpio[14] を使用した．Scorpio は特殊なプログラム依存グラフを用いるコードクローン検出ツールである．プログラム依存グラフとは，ノードで実行文，エッジで依存関係を表現したグラフであり，ソースコードから図 11 のように生成される．プログラム依存グラフを用いたコードクローン検出は，ギャップを含むコードクローン（文献 [9] 内では非連続コードクローン）を検出することができるという長所を持つ [9]．

Scorpio を用いて，ANTLR2.7.4 からギャップを含むコードクローンを検出し，その中から

メソッドの記述を大きく占めており，ギャップを複数含むコードクローンを調査し，それらを含む 2 つの類似メソッドを対象に決定した．対象としたメソッドは CppCodeGenerator クラスの genErrorHandler メソッド と，JavaCodeGenerator クラスの genErrorHandler メソッドである．

4.2 提案手法を用いたリファクタリング

CppCodeGenerator クラスの genErrorHandler メソッド と JavaCodeGenerator クラスの genErrorHandler メソッドを提案手法を実装したプラグインへの入力とした結果，表 1 に示す数の候補が提示された．

提示された候補の中から，分類 1 及び分類 3 の候補から 10 個ずつ，分類 6 の候補から 20 個を選び，実際にリファクタリングを行った．それぞれに必要だった操作の調査，及びリファクタリングの前後でソフトウェアの外部的動作に変化がないか確認を行った．

必要だった操作の調査

リファクタリングに必要だった操作を調査した．ただし，メソッドの抽出，メソッドの引き上げ，及び親クラスにおける抽象メソッドの定義は，Eclipse の既存の機能を用いることで行うことができ，また全てのリファクタリングにおいて行うため，これらの操作には含まない．調査する操作は以下のとおりである．

引数の変更（追加） 節 3.5 における条件 2 を満たしているコード片の場合，抽出後のメソッドの引数が一致するよう引数を追加する．

戻り値の変更 節 3.5 における条件 1 を満たす場合，戻り値の型，及び値を返す変数が共通となるように抽出前，または抽出後のコード片を修正する．

メソッドの作成 節 3.5 における条件 3 を満たしている場合，片方が抽出を行うコード片がないため，新たなメソッドを作成し，その呼出文を対応する位置に挿入する．

リファクタリングを行った結果，候補に対しそれぞれ必要な操作は表 2 のようになった．

表 1: ツールの実行結果

	分類 1	分類 2	分類 3	分類 4	分類 5	分類 6
候補の数	31	0	122	0	0	159

リファクタリング前後の動作の確認

それぞれのリファクタリングを行ったソフトウェアに対し，86 個のテストケースを用いたテストを行い，リファクタリングの前後において外部的動作に変化がないことを確認した．

4.3 考察

分類された候補を用いてリファクタリングを行った結果，それぞれに対して必要な操作の数から，回数の違い毎に正しく分類されていることが確認できた．さらに分類 1 の候補を用いたリファクタリングについては，表 2 のように，必要な操作がなかった．そのため容易に“ Template Method の形成 ”を行うことができたことから，記述を揃えることが容易な候補を絞り込むことができた．これらの結果から分類の有効性を確認でき，目的 2 を達成できた．しかし，分類 6 については，表 2 のように必要な操作の種類及び回数に差が生じているため，さらに多くの分類を考案する必要があると考えられる．さらに提示された候補の総数は 312 個であり，分類を行ったとしても多くなっている．そのため，これらの候補の中から有効性の高いものを特定する，または必要のないものを除外するためのメトリクスを考案する必要もあると考えられる．

リファクタリングの前後で外部的動作に変化がないことを確認できたため，提示された候補がリファクタリングに有効であると確認でき，目的 1 を達成できた．しかし候補を用いたリファクタリングの結果，親クラス（CodeGenerator クラス）に新たに定義した抽象メソッドを，Template Method パターンとは関係のない子クラスでもオーバーライドする必要がある構造になってしまった．また，CppCodeGenerator クラスと JavaCodeGenerator クラスの genErrorHandler メソッドだけでなく，CSharpCodeGenerator クラスの genErrorHandler メソッドも，ギャップとなる位置は異なるが，前者の 2 つのメソッドとギャップを含むコード

表 2: 必要だった操作

	分類 1	分類 3	分類 6
操作なし	10	0	0
引数の変更 1 回	0	10	0
引数の変更 2 回	0	0	4
引数の変更 1 回＋メソッドの作成 1 回	0	0	6
引数の変更 2 回＋メソッドの作成 1 回	0	0	1
引数の変更 1 回＋メソッドの作成 2 回	0	0	9

クローンとなっているため，これら 3 つのメソッド間において抽出箇所を検出できれば，さらに優れた構造になる可能性があると考えられる．

5　関連研究

Juillerat らの手法

Juillerat らは，Template Mehtod の形成をソースコードへ行う作業を自動化する手法を提案している [10]．Juillerat らの手法は，本手法と同様にソースコードから生成した抽象構文木を用いて，ギャップとなる部分木を検出する．しかし本手法がギャップとなる部分木を検出するために，抽象構文木の根から構造的に比較を行ったのに対し，Juillerat らの手法では，2 つの抽象構文木を深さ優先探索の帰りがけ順に探索することで作成した AST ノードの列に対して比較を行う．列の比較によって得たギャップとなるノードを，全て含むような最小の部分木を特定することで，ギャップとなる部分木を検出している．本手法の比較に比べ，抽象構文木の構造的な情報を失うが，高速度の比較を行うことが可能となっている．

メソッドの抽出においても本手法と異なる点がある．本手法では，節 3.3 で説明した条件を満たしたギャップである部分木が検出された場合，抽出する部分木列に隣り合う部分木を加え，抽出範囲を広げることで抽出可能な部分木列を検出している．比べて Juillerat らの手法では，ギャップとなる部分木が節 3.3 の条件を満たすそれぞれの場合にに対し以下の解決策を取っている．

条件 1 を満たす場合 互いに共通な代入文を特定し，片方に足りない代入文は，対応する位置に同じ変数に対する値が変化しない代入文を追加し，抽出する場合はそれらを 1 つずつ抽出することで，抽出範囲を変化させず抽出を行える状態に変化させている．

条件 2，または条件 3 を満たす場合 抽出したメソッドの戻り値に専用のクラスインスタンスを使用し，抽出後の記述にその戻り値によって break 文，continue 文，return 文に分岐する記述を加えることで，抽出範囲を変化させず抽出を行える状態に変化させている．

本手法は，範囲の拡大を行い，抽出する部分木列の候補を複数検出することで，開発者が適切だと考える候補を選ぶことの支援としているが，Juillerat らの手法は，全ての抽出を自動化することを目的としているため，範囲を変化させず抽出後の形を 1 つとしている．

6　まとめと今後の課題

本研究では，類似した 2 つのメソッドに対し，Template Method パターンを適用するリファクタリングの支援を行った．具体的には，ソースコードから生成した抽象構文木に基づいた比較による差分の検出，抽出することで検出した差分を分離することができる範囲の候補の検出，及びその提示を行う．抽出を行っても記述が揃わない場合があるが，抽出後の記述で何が異なっているかを基準に候補を分類することで，候補ごとにリファクタリングが容易であるか困難であるかを提示する．

この提案手法を，統合開発環境 Eclipse のプラグインツールとして実装し，実際に Java で記述されたソースコードを対象に適用実験を行い，また提示された候補を用いたリファクタリングを実際に行い，候補の有効性，およびその分類の有効性を確認した．

今後の課題としては，2 つのメソッド間での異なるユーザ定義名の対応付け，及び 3 つ以上の類似メソッドへの適用支援，の 2 つが考えられる．

本手法は，ユーザ定義名については，文字列が一致していない限りギャップとして扱う．しかしこの場合，ユーザ定義名が異なるが，共通の動作を含む 2 つの類似メソッドを対象としたリファクタリングを支援することが難しい．類似メソッド間でのユーザ定義名を，記述が異なる場合でも対応させ，その上で共通であるコード片とギャップであるコード片を把握することができれば，より多くの類似メソッドに対しリファクタリングの支援を行うことが可能となる．解決策としては，ユーザ定義名を正規化した状態でソースコードから生成した抽象構文木の比較を行い，その比較結果に基づいて，共通であるコード片におけるユーザ定義名の宣言，及び参照を順に番号付けをすることで対応付けを行う．この段階で対応関係が一致しないユーザ定義名，新たなギャップであるコード片を検出する要素とする．これらの操作によりユーザ定義名が異なる動作が類似したコード片を特定することができ，動作のみが異なるコード片を検出することができる．

また本手法は 3 つ以上の類似メソッドへの適用支援を行うことができない．しかし共通の親クラスを持つ 3 つ以上の子クラスが全て類似メソッドを持っている可能性もあり，これら全てに対し Template Method を適用することができれば，本手法を用いたリファクタリングを行った結果得られるソースコードよりも，優れた構造を持つソースコードを得ることが出来ると考えられる．3 つ以上の類似メソッド間における差分を含むコード片を特定することができれば，抽出可能であるコード片の候補の検出，及びその分類は現在の手法を応用することで可能であると考えられるため，3 つ以上の類似メソッド間における差分を含むコード片を特定する方法を考案することが，解決のための課題となる．

謝辞

本研究において，常に適切な御指導及び御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 教授に心より深く感謝いたします．

本研究において，随時適切な御指導及び御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 松下 誠 准教授に深く感謝いたします．

本研究において，逐次適切な御指導及び御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 石尾 隆 助教に深く感謝いたします．

本研究において，適時適切な御指導及び御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 早瀬 康裕 特任助教に深く感謝いたします．

本研究において，終始適切な御指導及び御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 吉田 則裕 氏に深く感謝いたします．

本研究において，数多くの御指導及び御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 譜久島 亮 氏，吉田 昌友 氏に深く感謝いたします．

本研究において，様々な御指導及び御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 斉藤 晃 氏，山田 吾郎 氏に深く感謝いたします．

最後に，その他様々な御指導，御助言等を頂い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研究室の皆様に深く感謝いたします．

参考文献

[1] ANTLR Parser Generator. `http://www.antlr.org/`.

[2] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of the International Conference on Software Engineering '98*, pp. 368–377, Kyoto, Japan, 1998.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluationof Clone Detection Tools. *IEEE Transaction on Software Engineering*, 31:804–818, 2007.

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

[5] M. Fowler. `http://refactoring.com/`.

[6] M. Fowler. *Refactoring: Inproving the Design of Existing Code*. Addison Wesley, 2000.

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[8] 肥後芳樹, 神谷年洋, 楠本真二, 井上克郎. コードクローンを対象としたリファクタリング支援環境. 電子情報通信学会論文誌, J88-D-I(2):186–195, 2005.

[9] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線 2009(ソフトウェアエンジニアリングシンポジウム 2009 予稿集), pp. 97–104, 2009.

[10] N. Juillerat and B. Hirsbrunner. Toward an Implementation of the "Form Template Method" Refactoring. In *Proc. of Seventh IEEE International Working Comference on Source Code Analysis and Manipulation*, Paris, France, 2007.

[11] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[12] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proc. of the International Symposium on Empirical Software Engineering 2004*, pp. 83–92, CA, USA, 2004.

[13] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[14] Scorpio. `http://sdl.ist.osaka-u.ac.jp/~higo/cgi-bin/moin.cgi/scorpio/`.

[15] R. B. Yates and B. R. Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

図 12: 類似文字列マッチングアルゴリズムの文字列比較表

付録

A　類似文字列マッチングアルゴリズム

類似文字列マッチングアルゴリズム [15] は，与えられた 2 つの文字列がどれだけ類似しているか，を計算するアルゴリズムである．片方の文字列に，1 文字を削除，1 文字を挿入，1 文字を置換，という 3 つの操作を最低何度行うことでもう片方と同一の文字列へと変化させられるか（編集距離）を調べることで，2 つの文字列の類似度を測ることができる．またこれらの編集距離を調べることで，2 つの文字列内の共通部分文字列，及びギャップである文字を特定することができる．

このアルゴリズムでは，2 つの文字列に対し図 12 のような表を生成する（文字列を記述している行，及び列は，実際には表には含まれていない）．

表の 1 行目には 0 を，1 列目には 0 から順に列の最後まで数字を格納する．2 行 2 列目から右，及び下のセルには，2 つの文字列それぞれ 1 文字ずつの比較結果によって決定する値を格納していく．1 行目及び 1 列目には既に全て値が格納されているので，2 つの文字列それぞれの先頭の文字の比較結果によって決定する値は 2 行 2 列目に格納される．値は以下の条件によって決定する．

同じ文字の場合 左上のセルの値を格納する

異なる文字の場合 左，上，左上のセルが格納している値のうち，最小の値に 1 を足した値を格納する．

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

図 13: 類似文字列マッチングアルゴリズムの文字列比較表の探索

上記の方法に従い，表の全てのセルに値を格納する．続いて，図 13 のように，表を探索する．探索する過程で，どう移動するかによって，2 つの文字列においてどの文字がギャップとなっているかを判断することができる．探索は以下の手順で行う．

- 最終列における，格納されている値が最も小さく，また行数が最も大きいセルから開始する．
- 左，上，左上のセルの中で，格納している値が最も小さいセルに移動する．複数のセルが最小値を格納している場合は，左上，左，上の順に優先し移動する．
- 1 行目，1 列目に到達した場合，探索を終了する．

表における，左の文字列を文字列 1，上の文字列を文字列 2 とする．探索の過程における移動方向によって，移動の視点であるセルと同じ行の文字列 1 の文字，同じ列の文字列 2 の文字がどのような関係であるかを以下のように判断できる．

左上に移動する場合 値が変化しなければ共通の文字であり，値が減少するならば互いに異なる文字である．文字列 1 の文字を文字列 2 の文字へ置換することで一致させることができる．

上に移動する場合 文字列 1 の文字のみが文字列 2 に対し余分に存在している．文字列 1 からこの文字を削除することで一致させることができる．

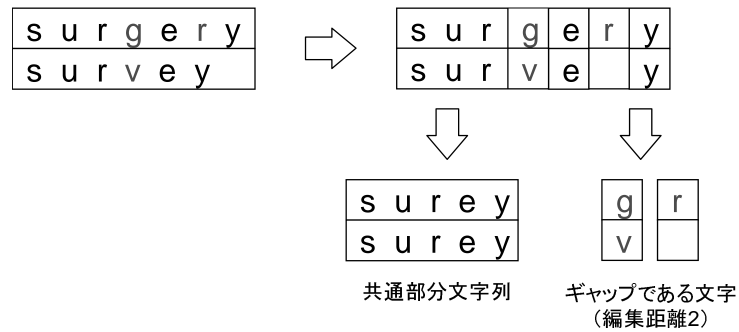

図 14: 類似文字列の共通部分文字列及びギャップである文字

左に移動する場合 文字列 2 の文字のみが文字列 1 に対し余分に存在している．文字列 1 にこの文字を挿入することで一致させることができる．

以上の操作で，2 つの文字間での編集距離を把握することができ，また共通部分文字列及びギャップである文字を特定することができる（図 14，参照）．