社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

ソースコード解析を対象としたコード前処理手法の集約

中島　望[†]　柗本　真佑[†]　楠本　真二[†]

† 大阪大学大学院情報科学研究科
〒 565–0871 大阪府吹田市山田丘 1–5
E-mail: †{n-nakajm,shinsuke,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソースコード解析の分野では，構文的なばらつきの排除を目的とした，ソースコードの前処理が広く用いられる．代表的な前処理としては空行やコメント行の削除，フォーマットの統一，変数名のトークンへの置き換え等が挙げられる．これらの前処理は多くの研究にとって共通のタスクでありながら，そのノウハウやツールはこれまで共有されてこなかった．そこで本研究では，「ソースコードの中立化」という概念を導入する．中立化はソースコードから多様性を排除し，ソースコード解析の結果に及ぼす影響を回避することを目的としている．また中立化において，入力は常にコンパイル可能なソースコードとし，中立化の前後でソースコードの振る舞いを変えないことを前提としている．この前提により，Unix や CI/CD 等で用いられているソフトウェアパイプラインを模倣した，前処理の連鎖的な実行が可能になる．本研究ではソースコードの中立化という概念のもと，既存研究で行われてきたソースコードの中立化手法を集約する．また，集約された中立化手法をもとに，実際のソースコード解析に導入できる中立化のためのツールを提案する．

キーワード　ソースコード解析，ソフトウェアリポジトリマイニング，ソースコードの前処理

1. はじめに

ソフトウェア工学において，ソースコードは重要な研究対象のひとつである [1] [2]．ソースコードを解析する目的として，ソフトウェアの定量的な品質評価 [3] や欠陥予測 [4]，既存のソフトウェアの再利用 [5] 等が挙げられる．特に近年，新たな研究分野としてソフトウェアリポジトリマイニング（Mining Software Repositories, MSR）が活発に行われている．MSR では，ソースコードを開発履歴の情報と紐付けて解析することが可能となる．

あらゆるプログラミング言語の記法は一種の柔軟性を内包しており，ソースコードにはその柔軟性に起因する構文的な多様性が含まれている．多様性としては繰り返し文を実装する際の for 文または while 文の選択，三項演算子または if-else 文による条件式の実装等が挙げられる．開発者らによって頻繁に議論されるコーディングスタイルの違いもまた，プログラミング言語の記法の柔軟性に起因する多様性の一種である．このような柔軟性は開発者に多様な実装手段を提示する一方で，ソースコードの解析結果には悪影響を及ぼす [6] [7]．ソースコードの多様性による解析への悪影響を回避するために，既存研究ではソースコードの前処理が行われてきた．前処理の手法としては，空行やコメント文の削除 [8]，フォーマットの統一 [9]，変数名のトークンへの置き換え [10] 等が広く知られている．Lincke らの研究 [11] では，ソフトウェアメトリクスの計測結果が，計測ツールによって異なることを示している．これは，計測ツールによってメトリクス計測の方針が異なることが原因とされている．ソースコードの前処理は，このような問題の回避にも効果的である．これらの前処理は多くの研究で共通して実施されているものの，それらのノウハウやツールは十分に共有されてこなかった．そのため，研究者が複数の前処理を行う場合には，適用したい前処理のためにツールや情報を収集し，各前処理を独立に行う必要があった．

本研究では，**ソースコードの中立化**という概念を提案する．**ソースコードの中立化**とは，記述の柔軟性に起因するソースコードの多様性を排除することで，ソースコードを中立な状態に変換し，解析に与える悪影響を回避する行為を指す．提案する中立化では，中立化前後のソースコードに対して，ソースコードの振る舞いの維持，及び構文的正しさの維持の 2 つの制約を満たすことを前提とする．この二つの制約条件により，研究者は一連の前処理の組み合わせをパイプライン処理に見立てて，容易に連続実行できるようになる．本稿では，**ソースコードの中立化**という概念のもと，既存研究で行われてきたソースコードの前処理を集約する．また，本研究では Java で記述されたソースコードの中立化処理を行うためのコマンドラインツール，**Neu4j** を提案する．Neu4j は独立に行われてきた中立化を統合し，様々な組み合わせや並び替えを容易に実現する．

2. 研究動機

本章では，研究動機の例を示す．図 1 に研究動機となるソー

A.java

```
1  import java.time.LocalDateTime;
2
3  public class ExampleA {
4
5      // output current time in 12-hour notation
6      public void run() {
7          LocalDateTime d = LocalDateTime.now();
8          int hour = d.getHour();
9          int min = d.getMinute();
10         String timeDiv = getAmOrPm(d);
11         String time = hour + ":" + min + " " + timeDiv;
12         System.out.println(time);
13     }
14
15     // return current time division, AM or PM
16     String getAmOrPm(LocalDateTime d) {
17         if (d.getHour() < 12) {
18             return "AM";
19         } else {
20             return "PM";
21         }
22     }
23 }
```

B.java

```
1  import java.time.LocalDateTime;
2
3  public class ExampleB {
4      public void run() {
5          LocalDateTime d = LocalDateTime.now();
6          System.out.println(d.getHour() + ":"
            + d.getMinute() + " " + getAmOrPm(d));
7      }
8      String getAmOrPm(LocalDateTime d) {
9          return d.getHour() < 12 ? "AM" : "PM";
10     }
11 }
```

The difference in unnecessary information

The difference in conditionals

The difference in granularities of statements

図 1　研究動機の例．二つのソースコードはどちらも同じ機能を実装しているが，記法に違いが表れている．

スコードを示す．図 1 に示される二つのソースコードは全く同じ機能を実装しているが，三種類の構文的な違いが現れている．

2.1　解析に不必要な記述の違い

図 1 において，緑色でハイライトされた行はコメント行と空行を指している．ソースコード解析において，コメント行や空行は事前に排除されることが多い．例として，ソフトウェアメトリクスの一つである論理 LOC（logical lines of code）が挙げられる．論理 LOC は，コメント行と空行を排除したソースコードの行数を意味する．ソースファイルの行数を示す物理 LOC は，規模を表す最も単純な指標として広く利用される．一方，コメント行や空行はソフトウェアの本質的な要素ではなく，解析には不必要とみなされることもある [12]．また，コメント行や空行の挿入有無や挿入頻度は開発者によって様々である．このため，物理 LOC は不必要にソースコード間の差異を発生させてしまう原因となる．そのような差異を無視してソフトウェアの規模を測るために，論理 LOC が用いられている．

2.2　条件式の記述の違い

図 1 中の青色のハイライトは，条件式の記述の違いを示している．どのようなプログラミング言語においても，条件式は非常に基本的かつ重要な構文の一つである．条件式を実装する上では，if-else 文，三項演算子，switch-case 文等様々な選択肢があり，どれを使用するかは開発者の好みに強く依存する．図 1 においても，その選択の違いが示されている．A.java の 17〜21 行目と B.java の 9 行目では，全く同じ振る舞いをする条件式が示されている．A.java が if-else 文を用いている一方で，B.java は三項演算子を用いている．どちらの実装方法でも同じ機能を実現可能だが，プログラム文の数は全くもって異なる．このような違いはメトリクス計測やコードクローン解析の結果に影響する [6] [13]．

2.3　プログラム文の粒度の違い

図 1 中の赤色のハイライトは，ソースコード間のプログラム文の粒度の違いを示している．一部の開発者は，一時変数を好んで利用する．一時変数を利用する目的は，メソッドの返り値の役割の明確化や，デバッグのしやすさ等様々である．一方で，一時変数を介せずに一行のプログラム文に処理を凝縮させる開発者も存在する．このような好みの違いによって，ソースコードには一行あたりのプログラム文の粒度の違いが生じる．図 1 中の A.java は B.java よりも細粒度で記述されている．A.java の 8 行目から 12 行目には，メソッド呼び出しで得られた返り値を結合し，一行の文字列を生成して出力する処理が記述されている．**hour** や **min** 等の変数が一時的に利用されている．一方で，B.java の 6 行目には A.java の一連の処理が一行で記されている．B.java は一時変数を利用しておらず，一行のプログラム文に情報が凝縮されている．そのため，同じ機能を実装しているにも関わらずソースコードの行数に大きな差が生まれている．ソースコード解析時には，このような粒度の不均衡が排除されるのが理想である．プログラム文の粒度を統一するための手法として，Higo ら [7] はソースコードの平坦化を提案している．平坦化は粒度が荒く複雑な一行のプログラム文を複数の単純なプログラム文に分解する．この平坦化はソースコード解析の結果に有効な手段であることが示されている．

3.　ソースコードの中立化

本章では，提案する**ソースコードの中立化**について定義し，中立化の例および流れを図 2 を用いて示す．以降では，ソースコードの中立化という言葉を単に中立化と記述する．

3.1　定　　義

本稿では，あるソースコードの多様性を排除し，一定の基準で揃える処理をソースコードの中立化と定義する．ソースコードの多様性とは，構文やコーディングスタイル等に生じるソースコード間の様々な違いを指す．この多様性は，いずれもプログラミング言語の記述の柔軟性に起因するものである．中立化の目的は，多様性がソースコード解析に与える悪影響を回避することにある．中立化はソースコード解析のために行われるこ

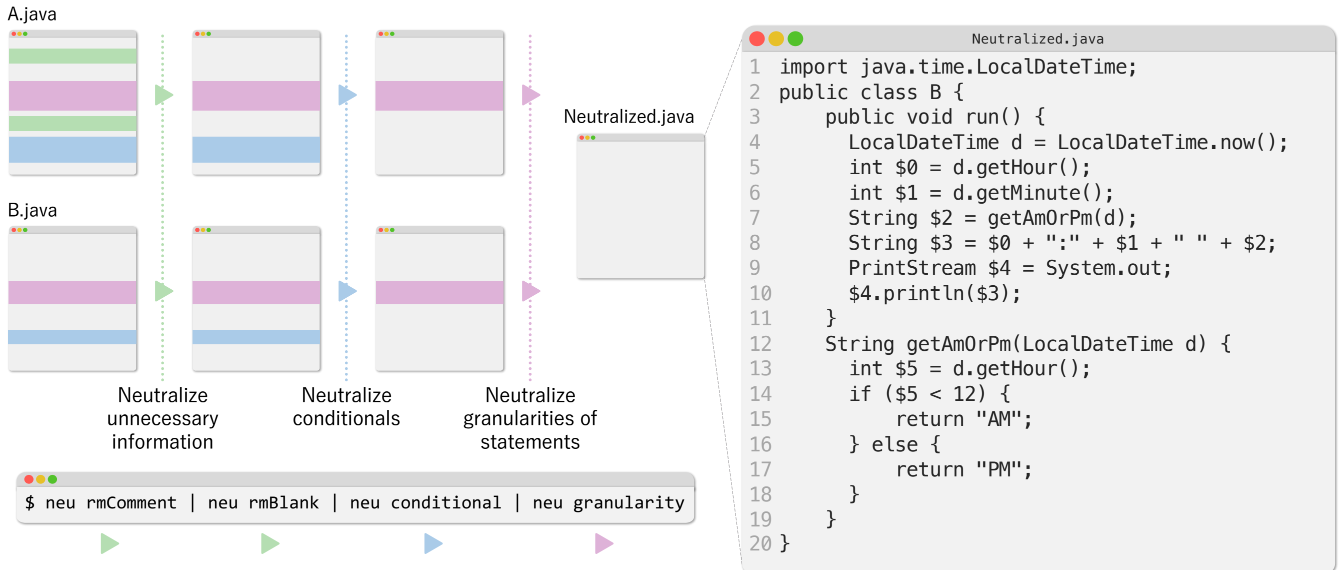

図 2　中立化の流れ. 二つのソースコードは図 1 に対応している.

とを前提としているため，理解性や可読性といったプログラム理解に必要な要素を考慮した変換は実施しない.

中立化に含まれる各変換処理を，**コンポーネント**と呼ぶ．全てのコンポーネントはソースコードを対象とした変換処理である．コンポーネントは，以下の二つの制約を満たすことを前提としている.

振る舞いの維持：コンポーネントによる変換の前後で，ソースコードの振る舞いが変わることはない.

コンパイル可能性：コンポーネントに入力されるソースコードは常にコンパイル可能なものである．またコンポーネントによって構文的な正しさが失われることはなく，常に構文的に正しい，コンパイル可能なソースコードが出力される.

この二つの制約を満たす変換処理のみがコンポーネントとして，中立化という概念のもとに集約されている.

3.2　特　　徴

前節で述べた通り，中立化に導入されているコンポーネントはコンパイル可能性を満たしたソースコードに対して処理を行う．またコンポーネントによって振る舞いが変えられることはなく，出力されるソースコードもコンパイル可能性を満たしている．これらの制約により，中立化の対象となるソースコードに対して，コンポーネントの適用順序や適用回数を制限する必要がなくなる．その結果として，中立化を行う場合，研究者は各コンポーネントを自由に組み合わせ，柔軟に並べ替えられるという利点を得ることができる．この中立化の特徴は，Unix や CI/CD（継続的インテグレーション/継続的デリバリー）等，多くのソフトウェアシステムで用いられているパイプラインの概念を模倣したものである．これまで個々に行われてきた様々な前処理をコンポーネントとして扱い，パイプラインでそれらを結合できる.

3.3　コンポーネントの例

中立化のコンポーネントにあたるソースコードの変換処理は，これまで様々な研究で実施されてきた．表 1 に Neu4j のコンポーネントの例を示す．コメント行や空行の削除は，多くの研究で用いられている [8] [14]．論理 LOC の計測時には，両行をソースコードから排除してファイルの行数をカウントする．フォーマットの統一も，よく知られるコンポーネントである [9] [15]．このようなコーディングスタイルの違いから生じる多様性には本質的な情報が含まれていない可能性が高く，解析前に取り除かれることが多い.

コーディングスタイルの違いだけでなく，構文の違いに着目したコンポーネントも存在する [6] [7] [13]．コードの平坦化はプログラム文の粒度を中立化するコンポーネントである．平坦化を行うと，複雑な一行のプログラム文が複数の単純なプログラム文に分解される．繰り返し文や条件式の統一もコンポーネントの一種である．開発者は繰り返し文を実装する際，for 文や拡張 for 文，while 文等の選択肢から場合や好みに応じた構文を選ぶ．条件式を実装する際にも if-else 文や三項演算子，switch-case 文といった様々な手段を検討できる．このような違いがソースコード解析に影響することは，既存研究で指摘されている [6] [13]．構文の違いを中立化するコンポーネントは，ソースコードの構造を変えることになる.

3.4　中立化の流れ

中立化の流れを示すために，例として図 2 を用いる．図 2 は図 1 に示された二つのソースコードを中立化する流れを示している．本例では，大きく分けて三段階でソースコードの多様性を排除する.

表 1　Neu4j のコンポーネントの例

カテゴリ	コンポーネント名
フォーマットの変換	コメントの削除
	空行の削除
	フォーマットの統一
構文の変換	コードの平坦化
	変数名のトークンへの置き換え
	条件式の統一

（1） 不要な情報の中立化

まず最初に，ソースコード中から解析に不要な情報を排除する．図 2 中に緑色で示された記号は，この中立化に該当する．コンポーネントとしては，コメント行の削除，空行の削除の二種類を利用する．中立化後，A.java と B.java に含まれていたコメント行・空行は削除される．

（2） 条件式の中立化

次に，条件式の中立化を行う．図 1 では条件式の実装にあたり，A.java では if-else 文を，B.java では三項演算子を用いていた．本例ではこれらを if-else 文に統一し条件式の違いを排除する．図 2 中に青色で示された記号は，この中立化に該当する．中立化の際には，条件式の統一を行うコンポーネントを利用する．この中立化により，B.java の三項演算子は if-else 文に変換される．

（3） 粒度の中立化

最後に，プログラム文の粒度の中立化を行う．A.java と B.java では，プログラム文の一行あたりの粒度に違いがあった．本例ではコードの平坦化を行うことで，この粒度のばらつきを中立化する．図 2 中に赤色で示された記号は，この中立化に該当する．コンポーネントとしては，ソースコードの平坦化を利用する．粒度の中立化後，A.java と B.java の粒度は均衡となる．

三段階の中立化を行った後に，A.java と B.java は図 2 の右側に示された Neutralized.java と全く同様の構造を持つソースコードへと変換される．ただし，本例では変数名や関数名に関する中立化は行っていないため，それらの違いは出力されるソースコードに残されたままとなる．本例に示された通り，様々なコンポーネントをつなぎあわせて中立化を行うことで，ソースコードの記法のばらつき徐々に統一されていく．

4. Neu4j

本章では，中立化を行うためのコマンドラインツール Neu4j について述べる．最初に Neu4j の概要を紹介し，その後現在の Neu4j の実装について述べる．

4.1 概　　要

前章で述べた通り，多くの研究でソースコードの前処理が行われている．また，その中には実際に前処理を行うツールを提供しているものも存在する．しかし，それらのツールは独立に提供されており，統合的に管理されていないのが現状である．既存の前処理ツールをコンポーネントとして統合し，一連の前処理として実行するために，本稿ではソースコードを中立化するためのツールとして **Neu4j** を提案する．Neu4j は Java のソースコードの中立化を目的としたコマンドラインツールである．ツール名は，Neutralization for Java の省略形である．

4.2 仕　　様

本節では，実際に Neu4j を用いて中立化を行う際に知る必要のある Neu4j の仕様について述べる．図 3 に Neu4j を用いた中立化の流れを示す．図 3 の上部では，Neu4j の構造を示している．また図の下部では，上部に示した Neu4j の構造に対応した，実際の実行コマンド例を示している．以降では図 3 を参照しながら，入出力ファイルの指定とコンポーネントの接続といった仕様について述べる．

コンポーネントの指定：Neu4j は `neu` コマンドを利用することで実行可能である．実行の際は，利用するコンポーネントの名前を第一引数に指定する．

コンポーネントの接続：中立化に導入されるコンポーネントは，振る舞いの維持とコンパイル可能性という二つの制約を満たしているものに限られていた．Neu4j の各コンポーネントについても，この二つの制約を満たしていることを前提としている．そのため，コンポーネントの実行順序や接続回数に制限はない．特定の入力ファイルに対して複数のコンポーネントを適用する場合には，図 3 の下部に示したコマンド例のように，Linux コマンドにおけるパイプラインでコンポーネントを結合する．

入出力の指定：Neu4j は，`-i` オプションで指定されたソースファイルもしくはソースディレクトリ下の全ソースファイルを中立化の対象として扱う．中立化されたソースコードは `-o` オプションで指定されたパスに出力される．ディレクトリが入力として指定された場合には，ディレクトリ構造を保ったまま中立化を行う．この出力パスは標準出力で共有されるため，パイプラインで結合した二番目以降のコンポーネントについては，入出力パスを指定する必要はない．

4.3 実　　装

現在，Neu4j の開発は試作段階にある．試作した Neu4j が持つコンポーネントとしては，空行の削除，コメント行の削除，フォーマットの統一，粒度の中立化がある．空行の削除とコメント行の削除によって，解析に不必要な情報を排除できる．フォーマットの統一は，コーディングスタイルの違いから生じる多様性を排除する．Neu4j におけるフォーマット統一の実装には，Eclipse の提供している CodeFormatter(注1)を利用している．また，Neu4j では Higo ら [7] によって提案されたプログラム文の粒度を中立化するコードの平坦化も導入している．

5. 適　用　例

中立化の適用例を示すために，本章では Neu4j を用いてソースコードを中立化する例を示す．実際の解析に近い例を示すために，二種類のソフトウェアメトリクスの計測を想定した中立化を実施する．

5.1 論理 LOC の計測

論理 LOC を計測する前に，ソースコード中のコメント行と空行を排除する必要がある．排除したファイルに対し，Linux コマンドの一つである `wc` コマンドを `-l` オプションとともに実行することで，論理 LOC を計測することができる．コメント行と空行の削除は，以下のコマンドで実行できる．

```
$ neu rmComment -i A.java -o out/A.java \
| neu rmBlank
```

コメント行の削除と空行の削除を行うコンポーネントをパイプ結

(注1)：https://github.com/eclipse/eclipse.jdt.core/blob/master/org.eclipse.jdt.core/formatter

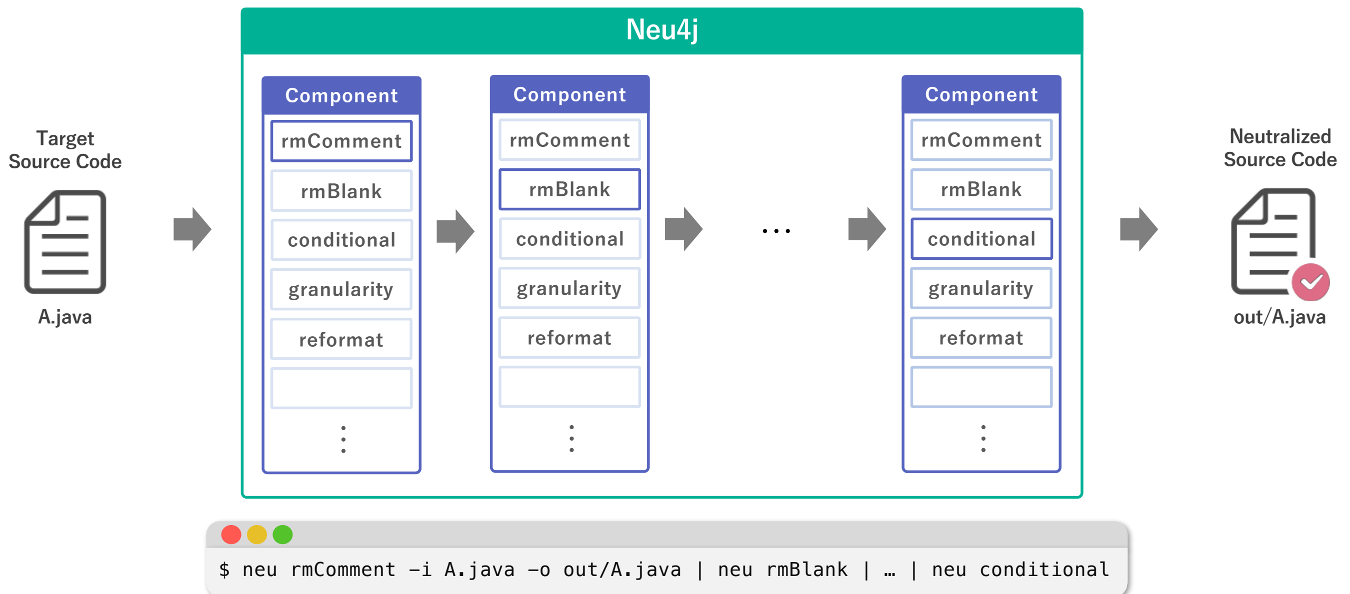

図 3　Neu4j を用いた中立化の流れ

合すれば良い．中立化されたソースコードに対して以下の通り wc コマンドを実行する．出力結果は論理 LOC となる．

```
$ wc -l out/A.java
```

上記のコマンドでは入出力に単一ファイルを指定したが，Neu4j ではディレクトリの指定も以下の通り可能である．

```
$ neu rmComment -i src/ -o out/ \
| neu rmBlank
```

ディレクトリ指定が可能になることで，研究者は個々のファイルを独立に処理する必要がなくなる．加え，使用するコンポーネントのインターフェースの違いを考慮する必要がなくなるため，コンポーネントに応じて入出力やオプションの指定方法を変更する必要もない．

5.2　細粒度 LOC の計測

本例では，中立化されたソースコードの論理 LOC を細粒度 LOC と定義する．ソースコードを平坦化することでプログラム文の粒度を揃えた場合の論理 LOC 等が該当する．細粒度 LOC の計測時には，論理 LOC と同様の処理に加えてソースコードを変換する必要がある．平坦化したソースコードの論理 LOC を計測する場合，以下のコマンドを実行することでソースコードを中立化できる．

```
$ neu rmComment -i A.java -o out/A.java \
| neu rmBlank | neu granularity
```

コメント行や空行の削除以外のコンポーネントに関しても，入出力のディレクトリ指定が可能である．また，平坦化以外にも様々なコンポーネントを組み合わせ，連続的に実行することができる．特定のディレクトリに対して条件式の統一と平坦化を行う場合，以下のコマンドを実行すれば良い．

```
$ neu rmComment -i src/ -o out/ \
| neu rmBlank | neu conditional \
| neu granularity
```

特に開発者らが個々に提供しているソースコードの変換ツールを前処理に利用する際，各ツールのインターフェースや制約を確認し，状況に応じて各前処理を個別に行ったり，各ファイルを単一で処理する必要がある．中立化は常にコンパイル可能なソースコードの入出力を前提としているため，このような違いを無視することができる．また，Neu4j は各前処理をコンポーネントとして含んでいるため，パイプラインで結合し，前処理を一括で行うことができる．

6.　関 連 研 究

ソースコード解析は，ソフトウェア工学において長年実施されてきた重要な研究分野の一つである [1] [2]．ソースコード解析の手法や実施の目的は様々である．ソフトウェアメトリクスの計測は，代表的な解析手法の一つである．メトリクスはソフトウェアの欠陥予測 [4] や品質の評価 [16] 等，様々な目的で利用されている．メトリクスの利用によって，定量的なソフトウェアの評価が可能になる [9]．コードクローンの検出もまた，広く実施されている解析手法である [17]．コードクローンの検出は既存ソフトウェアの再利用 [5] やリファクタリング [13] に貢献している．また近年，ソフトウェアリポジトリマイニングが非常に活発に行われている [18]．ソースコードを多種多様な開発履歴の情報と紐づけることが可能になり，より多くの情報を用いた解析が行えるようになった [19]．この他にも，近年では自然言語解析 [20] や深層学習 [21] 等の技術を用いたソースコード解析も行われている．

既存研究では，ソースコード中の多様性を排除し，中立化する手法が提案されている．Higo ら [7] は，ソースコードの平坦化を提案した．ソースコードの平坦化では，一行で記述された

複雑なプログラム文を，複数の単純なプログラム文に分解する．平坦化はソースコードの一行あたりの粒度を中立化していると言える．Qiao [6] は，同じ機能の実装にあたって，多様な実装方法が存在することを示している．例えば，繰り返し文を実装する際には for 文，拡張 for 文，while 文の利用が考えられる．三項演算子と if-else 文のどちらを利用するかというのもまた，多様な実装の例として挙げることができる．このような中立化の手法は個々に提案されており，これらを統合的に扱う研究は我々の知る限り存在していない．

7. おわりに

本研究では，ソースコード解析を対象としたコード前処理手法を集約するために，ソースコードの中立化という概念を提案した．中立化はソースコードの変換処理を行うことで多様性を排除し，解析に与える悪影響を回避することを目的としている．また，中立化を行うコマンドラインツール Neu4j を提案した．Neu4j を用いることで，様々な中立化手法をパイプライン接続できる．これにより，これまで独立で行われてきた前処理を一度に実行できるようになる．

今後の課題として，以下の点が挙げられる．現在，Neu4j は試作の段階であり，四種類のコンポーネントしか採用していない．うち三種類のコンポーネントはコーディングスタイルに関するものであり，ソースコードの構造を大きく変える変換処理ではない．本稿で挙げた構文に着目したコンポーネントを追加で採用することで，様々なコンポーネントの組み合わせが実現できる．特にソースコードの構文を変えるコンポーネントを複数実行する場合，実行順序によって中立化後のソースコードが異なる場合がある．コンポーネントの追加はより多くの中立化方法を実現し，Neu4j のソースコード解析への貢献度を高められると考えている．また，本研究では提案内容の実証的評価を実施していない．提案内容の妥当性を示すためにも，既存研究の再現実験等を行うことで実証的に評価する必要がある．

謝辞　本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

文　献

[1] D. Binkley, “Source Code Analysis: A Road Map,” Future of Software Engineering, pp.104–119, IEEE Computer Society, 2007.
[2] M. Harman, “Why Source Code Analysis and Manipulation Will Always Be Important,” Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation, pp.7–19, IEEE Computer Society, 2010.
[3] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, “Software Quality Analysis by Code Clones in Industrial Legacy Software,” Proceedings of the 8th International Symposium on Software Metrics, pp.87–, IEEE Computer Society, 2002.
[4] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction,” IEEE Trans. Softw. Eng., vol.31, no.10, pp.897–910, 2005.
[5] N. Kawamitsu, T. Ishio, T. Kanda, R.G. Kula, C. De Roover, and K. Inoue, “Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity,” Proceedings of IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp.305–314, 2014.
[6] Guo, Qiao, “Mining and Analysis of Control Structure Variant Clones”. Master’s thesis, Concordia University, 2015.
[7] Y. Higo and S. Kusumoto, “Flattening Code for Metrics Measurement and Analysis,” Proceedings of IEEE International Conference on Software Maintenance and Evolution, pp.494–498, 2017.
[8] J.A. Jones and M.J. Harrold, “Empirical Evaluation of the Tarantula Automatic Fault-localization Technique,” Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp.273–282, ACM, 2005.
[9] S. Ducasse, O. Nierstrasz, and M. Rieger, “On the Effectiveness of Clone Detection by String Matching: Research Articles,” J. Softw. Maint. Evol., vol.18, no.1, pp.37–58, 2006.
[10] 森崎修司，吉田則裕，肥後芳樹，楠本真二，井上克郎，佐々木健介，村上浩二，松井　恭，“コードクローン検索による類似不具合検出の実証的評価，” 電子情報通信学会論文誌. D, 情報・システム，vol.91，no.10，pp.2466–2477，2008.
[11] R. Lincke, J. Lundberg, and W. Löwe, “Comparing Software Metrics Tools,” Proceedings of the International Symposium on Software Testing and Analysis, pp.131–142, 2008.
[12] R. Humaira, K. Sakamoto, A. Ohashi, H. Washizaki, and Y. Fukazawa, “Towards a Unified Source Code Measurement Framework Supporting Multiple Programming Languages,” Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering, pp.480–485, 2012.
[13] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, “JDeodorant: Clone Refactoring,” Proceedings of the 38th International Conference on Software Engineering Companion, pp.613–616, ACM, 2016.
[14] B.S. Baker, “On Finding Duplication and Near-duplication in Large Software Systems,” Proceedings of the Second Working Conference on Reverse Engineering, pp.86–, IEEE Computer Society, 1995.
[15] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, “Finding file clones in FreeBSD Ports Collection,” Proceedings 7th IEEE Working Conference on Mining Software Repositories, pp.102–105, 2010.
[16] H. Barkmann, R. Lincke, and W. Lowe, “Quantitative Evaluation of Software Quality Metrics in Open-Source Projects,” Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops, pp.1067–1072, IEEE Computer Society, 2009.
[17] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code,” IEEE Transactions on Software Engineering, vol.28, no.7, pp.654–670, 2002.
[18] A.E. Hassan, “The road ahead for Mining Software Repositories,” 2008 Frontiers of Software Maintenance, pp.48–57, 2008.
[19] 門田暁人，伊原彰紀，松本健一，“『ソフトウェア工学の実証的アプローチ』シリーズ第 5 回ソフトウェアリポジトリマイニング，” コンピュータ ソフトウェア，vol.30，no.2，pp.2_52–2_65，2013.
[20] A. Hindle, E.T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the Naturalness of Software,” Proceedings of the 34th International Conference on Software Engineering, pp.837–847, IEEE Press, 2012.
[21] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Deep Learning Similarities from Different Representations of Source Code,” Proceedings of the 15th International Conference on Mining Software Repositories, pp.542–553, ACM, 2018.