ハッシュ値比較による Java バイトコードに含まれるライブラリの検出手法

矢野 裕貴[1,a)] Raula Gaikovina Kula[1,b)] 石尾 隆[1,c)] 井上 克郎[1,d)]

概要：ソフトウェア開発では，開発コストの削減のためにライブラリの利用が一般的に行われている．ソフトウェアの中には，ライブラリのコードをコピーして内部に含んでいるものが存在するが，このような再利用方法が取られている場合，利用中のライブラリに関する情報の管理が難しくなりがちである．そこで本研究では，Java のソフトウェアの内部に含まれるライブラリを検出する手法を提案する．Java においては，ソースコードではなく，ソースコードからコンパイラによって生成された，クラスファイルと呼ばれる Java バイトコードを含むファイルの再利用が主流である．そこで本手法では，バイトコードから抽出した情報をもとに計算したハッシュ値の比較に基づき，ライブラリの再利用を検出する．また，バイトコードから情報を抽出する際にパッケージ名に関する情報を取り除くことによって，パッケージリネームの検出を可能にした．

キーワード：ソフトウェア再利用，Java バイトコード，依存関係を含む JAR

1. まえがき

ソフトウェア開発において，外部のプロジェクトからコードを再利用することが一般的に行われている．また，ライブラリと呼ばれる再利用を行うためのコードも盛んに開発が行われており，数多くのオープンソースライブラリ (OSS ライブラリ) が配布されている．ライブラリの利用などによるコードの再利用によって，ソフトウェアの開発コストを大幅に削減することが可能になる一方で，再利用元にバグや脆弱性が含まれていた場合，開発中のプロジェクトにも同様の不具合を取り込んでしまう危険性が存在する．このような場合，利用の中止や修正版へのアップデートなどの対応を取ることで不具合を取り除かなくてはならない．そのためには再利用元のプロジェクト名とそのバージョン番号を正しく把握し，管理しておくことが重要となる．しかし，実際には再利用しているライブラリに関する情報が適切に管理されていないプロジェクトや，再利用しているライブラリが脆弱性を含んでいるプロジェクトが多数存在していることがわかっている [1].

再利用元がわからなくなってしまったコードの再利用元を特定する既存研究は数多く存在し，例えばソースコードが利用可能な場合には，コードの一部などを入力することによって一致するコード片を含むソースコードを検索する Inoue らの Ichi Tracker[2] や，再利用されたと思われるファイルを入力とし，ソースコードの類似度によって出自を高速に検索する川満らの手法 [3] などを利用することが可能である，

本研究の対象としている Java では，ソースコードではなくクラスファイルと呼ばれるバイトコードを再利用することが可能である．実際に，Heinemann らがオープンソースの Java のプロジェクトに対して行った調査によると，ソースコードの再利用よりもバイトコードの再利用が多く行われていることが判明している [4]．このような場合には，Davice らの Software Bertillonage[5] のような JAR (Java アーカイブ) の比較手法が出自検索に利用可能である．しかし，Java では様々なライブラリに由来する複数のファイルを，jar ファイルとしてまとめたものを配布することが一般的に行われている．このような場合には，どのファイルが再利用されたものなのか特定することが難しく，Davice らの手法は利用することができない．

複数のライブラリが含まれる jar ファイルから，バイトコードの比較に基づいて再利用元を特定する手法としては，Ishio らの Software Ingredients[6] が存在する．この手法を用いることで，例えば，Java のソフトウェアの内部か

1 大阪大学
Osaka University, Suita, Osaka 565-0871, Japan
a) y-yano@ist.osaka-u.ac.jp
b) raula-k@ist.osaka-u.ac.jp
c) ishio@ist.osaka-u.ac.jp
d) inoue@ist.osaka-u.ac.jp

ⓒ 2017 Information Processing Society of Japan

ら脆弱性が含まれるライブラリが検出された場合，利用をやめるなどの対処をとることが可能となる．しかし，この手法にはバイトコードの再利用を行う際に，パッケージ名が変更されていた場合には検出することができないという問題点がある．様々な理由によってパッケージのリネームは行われ，さらにそのようなコンポーネントは再利用元情報の管理が難しくなると考えられる．

本研究では，入力された Java ソフトウェアをあらかじめ作成しておいたデータベースと比較することによって，内部に含まれるライブラリを検出する手法を提案する．提案手法では，クラスファイル (Java バイトコードを含むファイル) ごとに特徴となるような情報を抽出し，それを用いて計算したハッシュ値を比較に用いる．ハッシュ値の計算にパッケージ名の情報を用いないことによって，パッケージがリネームされていても出自が同じクラスファイルであれば同じハッシュ値となるようにしている．それを用いて比較を行うことによってパッケージのリネームを検出することを可能とする．

以降，2 章では本研究の背景について述べる．3 章では提案手法について述べる．4 章ではオープンソースソフトウェアに含まれるライブラリの再利用を検出するケーススタディについて述べる．5 章では本研究のまとめと今後の課題を述べる．

2. 背景

2.1 再利用分析

Java ではライブラリによるコードの再利用が主流である．実際に，Bavota らの調査 [7] によって Java のプロジェクトが利用するライブラリの数は開発が進むにつれて増加していくことが判明している．ライブラリの利用の分析を行うことによって，効率的で安全なライブラリの利用が可能になり，ソフトウェア開発の助けになると考えられる．

Kula らはプロジェクトにおける利用中のライブラリのアップデート履歴を可視化する手法を提案した [8]．Maven リポジトリーにおけるライブラリの利用状況の統計情報と組み合わせて使用することで，アップデートを行うべきコンポーネントを特定することが可能になるとされている．

関連して我々の研究グループでは，ライブラリの利用状況の統計情報を組み合わせで検索可能なツールを提案した [9]．あるライブラリをアップデートしようとするときにはライブラリの後方互換性の問題により，他のライブラリが動作しなくなってしまう危険性がある．利用実績のある組み合わせを提示し，それに従って他のライブラリのアップデートも考慮することによって互換性による問題が発生するリスクを低減させることが可能になると考えられる．

これらの研究では，ライブラリの利用状況に関する情報を Maven というプロジェクト管理ツールの POM ファイルと呼ばれる設定ファイルから取得している．しかし，実際にはソースコードやバイナリをコピーすることで内部に取り込んでいるソフトウェア (ライブラリ) が数多く存在するため，設定ファイルからは取得不可能な再利用情報も考慮することが望ましい．そのため，ソフトウェアの内部に含まれるコードの情報を用いて，再利用の検出を行う研究と組み合わせることが必要であると考えられる．

再利用元の検出を行う研究としては，ソースコードが入手可能な場合には，コードの一部などを入力することによって一致するコード片を含むソースコードを検索する Inoue らの Ichi Tracker[2] や，LSH アルゴリズムを利用しソースファイル間の類似度を高速で推定する川満らのツール [3] などが再利用の分析に利用可能である．

ソースコードが利用できない場合には，バイナリファイルやその他の情報を用いて再利用分析を行う．Teyton らは jar ファイルに含まれるパッケージの名前を用いてライブラリ名を特定することによって，Java のソフトウェアにおける利用ライブラリの移行を検出する手法を提案している [10]．この手法を用いることでソフトウェア内部に含まれるライブラリ名の一覧の特定が可能であるが，バージョン番号までは特定することができない．

Davice らの Software Bertillonage[5] では，jar ファイル内部に含まれるクラスファイル (バイナリファイル) に含まれるシグネチャに関する情報を抽出し，比較を行うことによって，jar ファイル間の類似度を定義している．この手法を用いると，入力ソフトウェアと類似度が最も高いソフトウェア，つまり再利用元と考えられるソフトウェアの名前とバージョンを特定することが可能である．そのため，ライブラリを利用する際に jar ファイルの形のままで利用している場合には，そのバージョンを特定する方法として Software Bertillonage は優れている．しかし，全ての利用中のライブラリの jar ファイルを展開し，ソフトウェア固有のファイルとともに 1 つの jar ファイルにまとめているソフトウェアも存在する．このようなファイルは fat jar や jar with dependencies などと呼ばれ，Maven Assembly plug-in[*1] や eclipse[*2] の実行可能 jar ファイルを作成する機能などといった一般的な手段によって簡単に生成することが可能である．そのため，複数ライブラリに由来するクラスファイルが混ざった jar ファイルが出回ることは珍しくない．

Ishio らの Software Ingredients[6] はコンポーネントが jar ファイルにまとめられていない場合でも同様にソフトウェア内部に含まれるライブラリの一覧の検出が可能なツールである．Software Bertillonage と同様にクラスファイルから抽出したシグネチャの情報を用いて jar ファイルの比較を行い，一致するクラス数の割合を計算した後，その値が大きい順に優先度をつける貪欲法と呼ばれるアルゴ

[*1] http://maven.apache.org/plugins/maven-assembly-plugin/
[*2] https://eclipse.org/

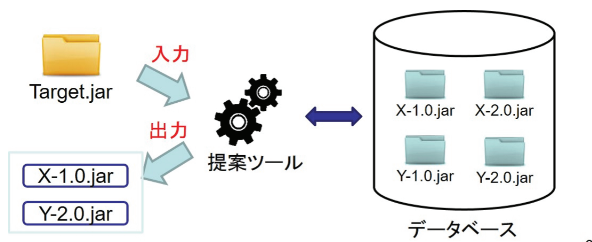

図 1　手法の概要

リズムによって再利用元を推定する．このツールは再利用元のライブラリからパッケージ名が変更されていた場合には検出することが不可能である．本研究では Software Ingredients で用いられていた方法をベースに，パッケージがリネームされていても再利用されたコンポーネントとして検出可能な手法を提案する．

2.2　パッケージのリネーム

ライブラリを利用するにあたって，そのパッケージ名を変更するような場合がある．実際に，Maven Shade Plugin[*3] や，Jar Jar Links[*4] といった，利用するライブラリに含まれているパッケージの名前を自動で変更できるツールも開発されている．パッケージ名のリネームによって，ライブラリのバージョンの競合によるプログラム動作の変化を防ぐことが可能となるが，利用中のライブラリが把握しづらくなることにより，ライブラリの管理に悪影響を及ぼす危険性が存在する．

3.　提案手法

本研究では，Java のバイトコードに含まれる外部ライブラリを検出する手法を提案する．手法の概要は図 1 のようになる．入力として与えられたソフトウェア (.jar ファイル) を，あらかじめ用意しておいたデータベース内に含まれるライブラリ (.jar ファイル) と比較することによって，入力されたソフトウェアの内部に含まれると推定されるライブラリの一覧を得る．また，どのクラスがどのライブラリから再利用されたかについての情報も得ることができる．

jar ファイルには，コンパイルされた複数の Java バイトコードやそれが使用する画像などのリソースが含まれるが，本手法では内部に含まれるクラスファイルのみを比較に利用する．また，直接クラスファイルを比較するのではなく，クラスファイル毎にハッシュ値を計算し，比較に用いることで計算量を削減している．本章では，手法の具体的な手順について説明する．

3.1　クラスファイル毎のハッシュ値の計算

本手法ではソフトウェアの内部に含まれるバイトコード

*3　https://maven.apache.org/plugins/maven-shade-plugin/
*4　https://code.google.com/archive/p/jarjar/

(. クラスファイル) の比較によって再利用の検出を行う．しかし，データベースに含まれる多数のライブラリを対象に，入力ソフトウェアとバイトコードの直接比較を行うと膨大な計算コストがかかる．そのため，バイトコードから得ることができる情報からハッシュ値を計算し，それを用いて比較を行うことで高速化を図っている．一般的に，MD5 や SHA-1 などによるファイルハッシュがファイルが一致するかどうかの判定に用いられている．しかし同じソースコードから生成されたクラスファイルであっても，コンパイル時の環境によって違うバイトコード列を持つことがあり，ファイルから直接計算されたハッシュ値による比較では正確に再利用を検出できない恐れがある．そのため，コンパイル環境の影響を受けないと思われる情報をクラスファイルに含まれるバイトコード列から抽出し，連結した文字列からハッシュ値の計算を行う．

ハッシュ値の計算に用いる情報は，Davice らの Software Bertillonage[5] や Ishio らの Software Ingredients[6] でも用いられていたクラス名や，メソッド名などの情報に加えて，Java バイトコード命令の算術，論理演算の回数を用いている．ただしこの際にパッケージ名に関する情報を使用しないようにすることによって，クラスファイルが属するパッケージ名が違っていても同じファイルであると判定されるようにしている．これによって，パッケージのリネームの検出を可能にしている．

具体的には以下の情報を連結した文字列からハッシュ値を計算している．

- クラス名
- 親クラス名
- 実装しているインターフェース (順不同)
- フィールド宣言 (順不同)
  各フィールド宣言に対しては以下の情報を利用する．
  – フィールド名
  – フィールドの型
- メソッド宣言 (順不同)
  クラス内で宣言されているメソッドについては以下の情報を利用する．
  – メソッド名
  – アクセス修飾子
  – 引数の型
  – 返り値の型
  – メソッド呼び出し命令 (順不同)
  – 算術，論理演算命令の回数

クラスファイルごとに計算されたハッシュ値が一致する場合，すなわち上に示したような情報がすべて一致するファイル同士を，本手法では同じクラスファイルであるとして扱う．

jar ファイル内部に含まれるクラスファイル毎にハッシュ値を計算した結果，ハッシュ値の多重集合として jar ファ

ⓒ 2017 Information Processing Society of Japan

イルを扱うことができる．ここで集合ではなく多重集合としているのは，パッケージ名のみが異なるクラスファイルが同一 jar ファイル内に含まれているような場合に，ハッシュ値の衝突が起こりうるからである．衝突が起こった場合には要素が重複して存在することになる．

以降，本論文で扱う全ての集合は多重集合であるとする．また，以降ハッシュ値の多重集合をハッシュ値集合と表記することがある．

3.2 再利用元の推定

前節の方法で計算したハッシュ値の多重集合を用いて，入力された jar ファイルとデータベース内の jar ファイルを比較することによって再利用元を推定する．基本的に，ライブラリの再利用は jar ファイル単位でまとめて行われることが多い．また，ハッシュ値集合の固有性によるライブラリ名とバージョンの特定精度も高くなると考えられる．そのため，本手法では完全な形でコピーされたライブラリを優先的に検出する．

3.2.1 再利用元推定アルゴリズムの概要

再利用元を推定するアルゴリズムを Algorithm1 に示す．このアルゴリズムは検出対象となるソフトウェア (jar ファイル) t と比較対象となる複数のライブラリ (jar ファイル) を含むデータベース R を入力として，R から t に再利用されたと推定されるライブラリ群と，そのクラスファイルのハッシュ値の一覧を出力として得る．なお，入力データベースにおいて，ハッシュ値集合の全要素が完全に一致するようなライブラリの組はあらかじめ取り除かれているとする．また，Classes(x) は x の内部に含まれるクラスファイルのハッシュ値集合を返す関数である．x が複数の jar ファイルからなる場合にはそれらの和集合を返す．

このアルゴリズムは以下の 2 ステップの繰り返しによって構成される．

(1) 共通部分の計算による再利用元候補の決定 (4-14 行目)
(2) 再利用元の確定 (15-17 行目)

以降，各ステップについて説明する．

3.2.2 共通部分の計算による再利用元候補の決定

このステップではまず，入力 t が持つ再利用元が確定していないクラスファイルに対応するハッシュ値集合 A とライブラリが持つハッシュ値集合との比較によって，R に含まれる $r_1, r_2, ..., r_n$ それぞれに対して A との共通部分 $r'1, r'2, ..., r'_n$ を得る．この共通部分を用いて，各ライブラリが持つクラスファイルのうち A との共通部分がどれだけあるかの割合（以降，オーバーラップ値と呼ぶ) を計算する．オーバーラップ値は，

$$\frac{|A \cap r|}{|r|} = \frac{|r'|}{|r|}$$

によって得られ，ライブラリ r が持つクラスファイルのうち，t に含まれるクラスファイルとハッシュ値が一致する，

Algorithm 1 再利用元の推定アルゴリズム

INPUT: t : target jar file, $R = \{r_1, r_2, ..., r_n\}$: Database(Set of Library)
OUTPUT: Result: Subset of Repository(and overlapped classes)

1: $A \leftarrow Classes(t)$ ▷ 入力ファイルに含まれているクラスで再利用元が未確定のもの
2: $Result \leftarrow \{\}$
3: $i \in [i, |R|]$, $r'_i = r$ ▷ r_i における A との共通部分を入れる変数
4: **loop**
5: **for** $i = 1$ to $|R|$ **do**
6: **if** $r'_i \in Result$ **then**
7: **continue**
8: **end if**
9: $r'_i \leftarrow A \cap \mathrm{Classes}(r_i)$
10: **end for**
11: $m \leftarrow \max_{i \in [i,|R|]} \frac{|r'_i|}{|r_i|}$
12: **if** $m < threshold$ **then**
13: **break** ▷ オーバーラップの最大値が閾値以下ならアルゴリズムを終了する
14: **end if**
15: $R'_{max} \leftarrow \left\{ r'_n | \frac{|r'_n|}{|r_n|} = m \right\}$
16: $J \subseteq R'_{max} \wedge \mathrm{Classes}(J) \subseteq A$ を満たし，$|\mathrm{Classes}(J)|$ が最大となるような J を選択
17: $Result \leftarrow Result \cup J$
18: $A \leftarrow A \setminus J$
19: **end loop**
20: **return** $Result$

つまり再利用された可能性のあるクラスファイルの数の割合を表す．

データベース内の全ライブラリに対してオーバーラップ値の計算が終わった後，オーバーラップ値が最大のライブラリを全て選択し，次のステップに入力することで再利用元を確定する．ただし，最大のオーバーラップ値が閾値よりも小さくなった場合，データベース内に再利用されたライブラリは残っていないと判定し，結果 $Result$ を出力し，アルゴリズムを終了する．

例えば，図 2 のような入力がなされた場合を考える．図では，データベース内のライブラリに含まれるハッシュ値のうち，赤く表示されている部分が入力ファイル Target.jar との共通部分となっている．この場合に Target.jar に対するデータベース内のライブラリのオーバーラップ値を計算すると，A-1.0.jar の場合は要素数 2 に対して Target.jar との共通要素数が 2 であるため $2/2 = 1.0$，A-2.0.jar の場合は要素数 2 に対して共通要素数が 1 であるため $1/2 = 0.50$ となる．同様に計算すると，C-1.0.jar，B-1.0.jar，D-1.0.jar に対しては 1.0，B-2.0.jar に対しては 0.67 が得られる．

その後，各ライブラリに対して計算されたオーバーラップ値が最大のものを選択する．オーバーラップ値 1.0 が最大となるため，A-1.0.jar，C-1.0.jar，B-1.0.jar，D-1.0.jar が再利用元の候補として次のステップに入力される．

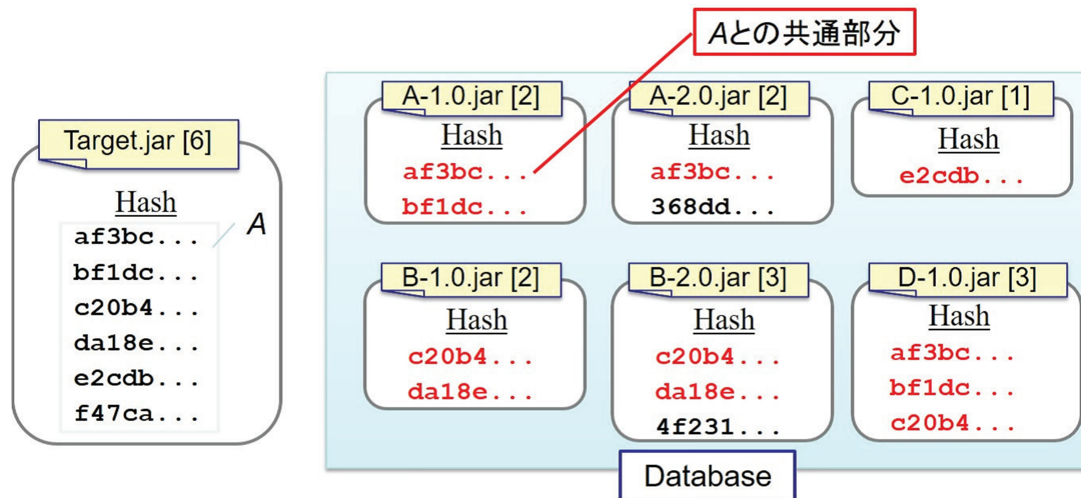

図 2　アルゴリズムへの入力例

3.2.3　再利用元の確定

このステップでは，前ステップによって得られた最大のオーバーラップ値を持つライブラリを候補とし，どれが再利用が行われたかの推定を行う．本手法では，入力 jar ファイルに含まれる 1 つのクラスファイルは，1 つのライブラリからのみ再利用されたものであるとしている．そのため，あるクラスファイルに対して同じハッシュ値のクラスファイルを持つライブラリが候補内に複数存在する場合には，再利用元となったものがどれなのかを特定する必要がある．例えば，図 3 のように，オーバーラップ値 1.0 を持つ 4 つのライブラリが再利用元候補であったとする．これらのライブラリ中で，ハッシュ値'af3bc...' を持つライブラリが A-1.0.jar と D-1.0.jar の 2 つあるのに対して，Target.jar には同じハッシュ値が 1 つしか含まれていない．このような場合に，ハッシュ値'af3bc...' に対応するクラスファイルが A-1.0.jar と D-1.0.jar のどちらから再利用されたかを特定する必要がある．

本ステップでは入力ソフトウェアに含まれるクラスファイルそれぞれを 1 つの再利用元に対応付けるため，Algorithm1 の 9 行目の条件の通り，各候補ライブラリの中から，A との共通部分の和が A の部分集合となるようなライブラリの組み合わせに対して探索を行う．その際，本手法においては，完全に近い形で再利用されているライブラリが再利用元であるとしているため，前述した条件のもとで要素数の合計が最大となるようなライブラリの組み合わせを求める組み合わせ最適化問題を解くことによって再利用元の推定を行う．このステップは以下のような手順に分けることができる．

(1) 候補内で固有のハッシュ値しか持たないものを再利用元として確定

基本的には条件を満たす組み合わせを全通り探索することによって要素数が最大となるものを特定するのだが，計算量削減のために，他のライブラリの選択に影響を与えないライブラリを固定で選択するようにする．具体的には，候補内で固有のハッシュ値のみが A との共通部分になっているようなライブラリを再利用元として確定する．

(2) 条件を満たすライブラリの組み合わせを深さ優先探索によって求める

前手順によって確定されたライブラリと合わせて，A との共通部分の和が A の部分集合とであるという条件を満たすようなライブラリの組み合わせを深さ優先探索によって全通り列挙する．深さ優先探索とは、あるノードから子のないノードに行き着くまで進んでいった後，最も近くの探索の終わっていないノードまで戻ることを繰り返すような探索方法である．この場合にはノードが選択されたライブラリとなり，その子ノードは共通部分の和が A の部分集合となるという条件のもとで選択可能なライブラリとなる．

(3) 再利用元の確定

全手順によって求めたライブラリの組み合わせの中から，含まれるハッシュ値の要素数が最大となるような組み合わせを選択する．その後，選択したライブラリ中に含まれるハッシュ値を A から取り除き，ステップ 1 の再利用元候補の決定に戻る．

図 3 は，本手法に図 2 のような入力がなされた場合の，1 回目のループにおける再利用元が確定していないクラスファイルに対応するハッシュ値の多重集合 A と，ステップ 1 で求められた再利用元候補の図となっている．再利用元候補となっている 4 つのライブラリはオーバーラップ値が 1.0 であるため，全要素が入力ソフトウェアとの共通部分となる．このような場合の検出例を示す．

まず，これら 4 つのライブラリを比較すると，C-1.0.jar は候補内の他のライブラリには含まれていない，ハッシュ

ⓒ 2017 Information Processing Society of Japan

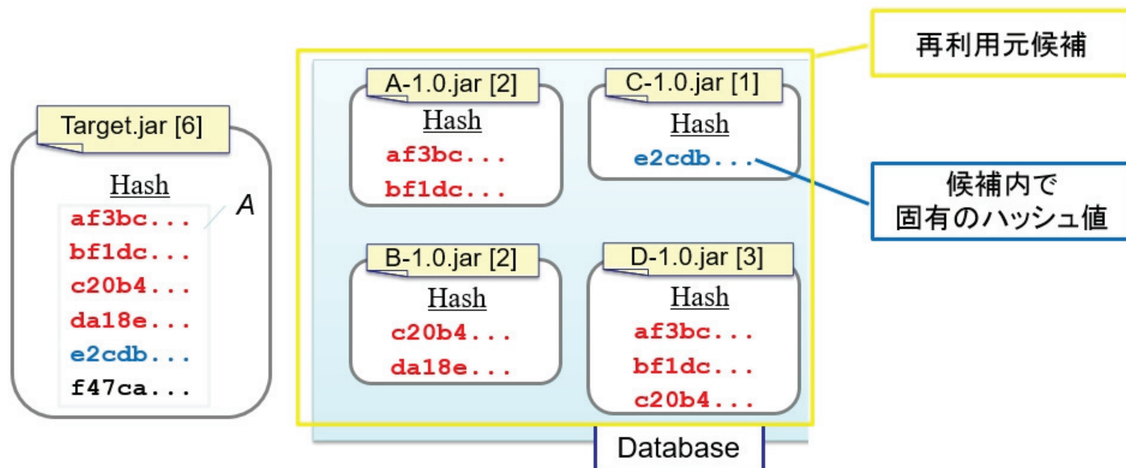

図 3　ループ 1 回目の再利用元候補

値'e2cdb' のクラスファイル 1 つのみを持っていることがわかる．そのため，Target.jar には C-1.0.jar が再利用されていると判断し，検出結果として確定する．

次に条件を満たすライブラリの組み合わせを列挙する．その結果，条件を満たし，要素数の和が 5 で最大となるようなライブラリの組み合わせ {C-1.0,A-1.0,B-1.0} が特定される．これによって，A に含まれる 6 つのクラスファイルのうち 5 つの再利用元が特定されたため，対応するハッシュ値を取り除く．

その後はステップ 1 の再利用元候補決定に戻ることになる．2 回目のループで A に含まれるのは，ハッシュ値'f47ca...' のみとなるが，同じハッシュ値のクラスファイルを持つライブラリはデータベース内に存在しない．そのため，各ライブラリのオーバーラップ値は 0 となり，アルゴリズムを終了する．以上により，検出結果として A.1.0.jar の 2 つのクラスファイル，B-1.0.jar の 2 つのクラスファイル C-1.0.jar の 1 つのクラスファイルが得られる．

4. ケーススタディ

本章では，提案手法のツールがパッケージのリネームを検出できるのかどうか調べるために，実際にライブラリを再利用する際にパッケージのリネームを行っている java のソフトウェアに対して提案手法を適用するとどのような結果が得られるかを調べた．

4.1 調査対象

Elastic Search[*5] のバージョン 0.90.5 に対して調査を行った．SoftwareIngredients にこのライブラリの jar ファイルを入力した結果，ライブラリの再利用は検出されなかった．

Elastic Search0.90.5 では，プロジェクトに関する設定などが記述されているファイル (pom.xml) 中の記述から，Maven Shade Plugin を用いて 9 つのライブラリに含まれるファイルを内部に展開して利用していることが判明している．それらのライブラリが使用していると思われるパッケージ名の変更設定も記述されていた．また，生成する Jar に含めるクラスファイルに関する設定である，MinimizeJar が使用されていた．

Maven Shade Plugin では，MinimizeJar オプションによって，動作が可能な最小の Jar ファイルを生成することが可能である．通常は生成 Jar ファイル中に，選択したライブラリに含まれる全てのクラスファイルがコピーされるが，このオプションを使用した場合，実際に使用されているクラスファイルのみを生成 Jar ファイルに含めるような仕組みになっている．そのため，Elastic Search の jar ファイル中には，9 つのライブラリに含まれるクラスファイルの一部が含まれていることになる．

再利用されているライブラリの一覧とそのパッケージ名を表 1 に示した．なお，これらのライブラリのバージョンに関する記述は発見することができなかった．

4.2 結果

提案手法を実装したツールにおいて，検索終了の閾値を 0.1 に設定し，elasticsearch-0.90.5.jar に対して内部に含まれるライブラリの検出を行った．結果として出力されたライブラリのリストを表 2 に示す．表 2 の各列は左から順に，検出されたライブラリ名，バージョン，含まれるクラスファイル数，その中で入力ソフトウェア内に再利用されていると判定されたクラスファイルの数，にそれぞれ対応している．バージョン番号については，ハッシュ値集合が等しいバージョンが複数ある場合には複数の候補を示している．例えば，reader-files のバージョン 1.1.2 には，31 個ハッシュ値集合が等しいバージョンが存在するため，表中にそれを記載している．

*5　https://www.elastic.co/products/elasticsearch

表 1　Elastic Search 0.90.5 に含まれるライブラリ

ライブラリ名	変更前パッケージ名	変更後パッケージ名
guava	com.google.common	org.elasticsearch.common
trove4j	gnu.trove	org.elasticsearch.common.trove
mvel2	org.mvel2	org.elasticsearch.common.mvel2
jackson-core	com.fasterxml.jackson	org.elasticsearch.common.jackson
jackson-dataformat-smile	com.fasterxml.jackson	org.elasticsearch.common.jackson
jackson-dataformat-yaml	com.fasterxml.jackson	org.elasticsearch.common.jackson
joda-time	org.joda	org.elasticsearch.common.joda
netty	org.jboss.netty	org.elasticsearch.common.netty
compress-lzf	com.ning.compress	org.elasticsearch.common.compress

表 2　Elastic Search 0.90.5 から検出されたライブラリ一覧

ライブラリ名	バージョン	クラス数	検出クラス数
guava-annotations	r03	4	4
guice-multibindings	2.0	4	4
guice-annotations	2.0.1	10	10
joda-time	2.3	157	144
jackson-core	2.2.3	69	63
mvel2	2.1.5.Final	349	253
guice-assisted-inject	2.0	7	5
guice	2.0-no_aop	184	123
jackson-dataformat-smile	2.2.2	12	8
	2.2.3		
jackson-dataformat-yaml	2.2.2	112	70
reader-files	1.1.2 *and 31 versions*	2	1
hadoop-job-builder	1.0	6	3
guava	15.0	453	205
	15.0-rc1		
	15.0-cdi1.0		
jsr166y	1.7.0	9	4
netty	3.7.0	546	239
compress-lzf	0.9.6	26	10
jellydoc-annotations	1.0 *and 4 versions*	3	1
trove4j	3.0.3	691	98

Elastic Search　0.90.5 において再利用されたとされている 9 つのライブラリは，表 2 中に太字で示した通りすべて検出結果に表れていることがわかる．Trove4j などは，ライブラリが持つクラスファイル数に対して検出されたクラスファイル数が少なくなっている．しかし，入力 jar ファイル中のディレクトリ org/elasticsearch/common/trove に含まれるクラスファイル数を確認した結果，103 個のファイルが含まれていることが確認できた．そのため，103 個のうち 98 個のクラスファイルを再利用元に対応付けることができており，再利用されたものの大部分を検出することに成功しているといえる．このディレクトリに含まれる残り 5 ファイルに関しては再利用元が特定できなかった．原因としては，再利用された trove4j のバージョンがデータベース内に含まれていなかったことが考えられる．

各ライブラリについて提案ツールによってバージョン番号が示されているが，全てのクラスファイルが再利用されたと判定されたものが存在しないため，これらのバージョン番号は一致するクラスファイルの数を元にデータベース内で一番近いものを選択した推定値となる．joda-time や jackson-core は大部分のクラスファイルが一致しているため比較的バージョン番号の特定精度も高いと思われるが，trove4j のようにライブラリ中のクラスファイルのうちのごく一部しか一致していないものは正しいバージョン番号が特定されている可能性は低い．

検出結果として，再利用されたとの記述がなされていた 9 個のライブラリの他に，guice とその関連ライブラリが多く表れている．これらのライブラリを比較対象から除外して再検索を行うと，guice から再利用されていたと判定されていたクラスファイルの再利用元が見つからなくなった．つまり，guice が設定ファイル中に記述されていた 9 個のライブラリに含まれていた可能性は低い．そのため，このライブラリは設定ファイル中には記載されていなかっ

たが，実際には再利用が行われている可能性が高い．

このケーススタディによって，誤検出が行われるような例も発見することができた．全てのクラスファイルが一致するとして guava-annotation が検出されているが，これは guava の一部機能のみを持つライブラリである．提案手法ではクラスファイルの一致する割合によって検出の優先度を定めているため，今回のようにあるライブラリの一部のみの再利用が行われている場合，特定機能のみが実装されているクラスファイル数の少ないライブラリがデータベースに登録されていた場合に，誤検出してしまうことが判明した．

5. まとめと今後の課題

本研究では，Java バイトコードから抽出した情報から計算したハッシュ値の比較によって，Java のソフトウェア内部に含まれるライブラリの再利用を検出する手法を提案した．提案手法では，バイトコードの比較を行う際にパッケージ名に関する情報を取り除くことによって，パッケージのリネームが行われていた場合でも再利用の検出を可能にした．

また，提案手法を実装したツールに複数個のライブラリを含む jar ファイルを入力することによって，再利用元を特定することが可能であるかどうか，既存研究との比較実験を行った．その結果，検出精度が向上していることが確認できた．一方で，検索にかかる時間については既存研究よりも増加していることが判明した．

今後の課題としては，検出アルゴリズムの改善が挙げられる．提案手法では，一致するクラスファイルの割合が大きいライブラリを再利用元として優先的に検出するアルゴリズムを採用している．しかし，実際に jar ファイルを生成する際には，ファイルサイズの削減のためにライブラリの一部のみを再利用しているような例が存在する．その結果，あるライブラリの一部機能のみを実装したライブラリなどの，規模の小さいライブラリが誤って検索結果として判定されてしまうような例が見られた．検出を行う際にライブラリ間の包含関係を考慮することによってこのような誤検出を減らすことが可能となる可能性がある．

謝辞 本研究は JSPS 科研費（課題番号 JP25220003，JP26280021）の助成を受けたものです．

参考文献

[1] Xia, P., Matsushita, M., Yoshida, N. and Inoue, K.: Studying Reuse of Out-dated Third-party Code in Open Source Projects, *JSSST Computer Software*, Vol. 30, No. 4, pp. 98–104 (2013).

[2] Inoue, K., Sasaki, Y., Xia, P. and Manabe, Y.: Where does this code come from and where does it go? – Integrated code history tracker for open source systems –, *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*, pp. 331–341 (online), DOI: 10.1109/ICSE.2012.6227181 (2012).

[3] 川満直弘， 石尾隆，井上克郎ほか：LSH アルゴリズムを利用した類似ソースコードの検索，研究報告ソフトウェア工学 (SE)，Vol. 2016, No. 7, pp. 1–8 (2016).

[4] Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B. and Irlbeck, M.: On the Extent and Nature of Software Reuse in Open Source Java Projects, *Proceedings of the 12th International Conference on Software Reuse*, Lecture Notes in Computer Science, Vol. 6727, pp. 207–222 (2011).

[5] Davies, J., German, D. M., Godfrey, M. W. and Hindle, A.: Software Bertillonage: Finding the Provenance of an Entity, *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 183–192 (2011).

[6] Ishio, T., Kula, R. G., Kanda, T., German, D. M. and Inoue, K.: Software ingredients: detection of third-party component reuse in Java software release, *Proceedings of the 13th International Conference on Mining Software Repositories*, ACM, pp. 339–350 (2016).

[7] Bavota, G., Canfora, G., Penta, M. D., Oliveto, R. and Panichella, S.: The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache, *Proceedings of the 29th IEEE International Conference on Software Maintenance*, pp. 280–289 (online), DOI: 10.1109/ICSM.2013.39 (2013).

[8] Kula, R. G., De Roover, C., German, D., Ishio, T. and Inoue, K.: Visualizing the Evolution of Systems and their Library Dependencies, *Proceedings of the 2nd IEEE Working Conference on Software Visualization*, pp. 127–136 (2014).

[9] Yano, Y., Kula, R. G., Ishio, T. and Inoue, K.: VerXCombo: An interactive data visualization of popular library version combinations, *Proceedings of the 23rd IEEE International Conference on Program Comprehension*, pp. 291–294 (2015).

[10] Teyton, C., Falleri, J.-R., Palyart, M. and Blanc, X.: A study of library migrations in Java, *Journal of Software: Evolution and Process*, Vol. 26, No. 11, pp. 1030–1052 (2014).

ⓒ 2017 Information Processing Society of Japan