THE IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS (JAPANESE EDITION)

電子情報通信学会 論文誌 D 情報・システム

VOL. J102-D NO. 11
NOVEMBER 2019

情報・システムソサイエティ

一般社団法人 電子情報通信学会
THE INFORMATION AND SYSTEMS SOCIETY
THE INSTITUTE OF ELECTRONICS, INFORMATION AND COMMUNICATION ENGINEERS

論　文

メソッドレベルセマンティックバージョニングの提案と評価

林　純一[†a)]　肥後　芳樹[†]　楠本　真二[†]

Proposition and Evaluation of Method-level Semantic Versioning

Junichi HAYASHI[†a)], Yoshiki HIGO[†], and Shinji KUSUMOTO[†]

あらまし　ソフトウェア開発において利用されるライブラリには，機能追加や改善，バグ修正などによる時系列的変化を区別するためのバージョンが付与される．バージョニング手法に後方互換性の有無を反映させるセマンティックバージョニングと呼ばれる手法がある．この手法でライブラリに付与されたバージョンからライブラリの後方互換性の有無は判断できるが，API ごとの後方互換性の有無は判断できない．そこで本研究では，メソッド単位でセマンティックバージョニングを行う手法を提案し，提案手法を用いて Java 言語のメソッドに自動でバージョニングを行うツールを作成した．本手法によりメソッド単位でバージョニングを行うことで，メソッドごとの後方互換性の有無を自動で判断できるようになる．更に，提案手法及び作成したツールの有用性を示すために，後方互換性のない変更が行われたメソッドを検出する実験を行い，既存研究との差異を調査した．その結果，提案手法を用いることにより，既存研究では検出できなかった変更を検出できることが分かった．

キーワード　メソッドレベルセマンティックバージョニング，バージョン管理システム，ソフトウェアライブラリ，API

1. まえがき

ソフトウェア開発において利用されるライブラリには，機能の追加や改善あるいはバグ修正が行われる．このような時系列的変化を区別するために「バージョン」が与えられる．このバージョンを決定する手法としてセマンティックバージョニング（Semantic Versioning）という，一つのバージョンを三つ組の整数を用いて表す方法が提唱されている [1]．

セマンティックバージョニングでは，公開されている API（Application Programming Interface）に行われた変更に後方互換性があるかどうかをバージョンに反映させる．Raemaekers らは既存の Java ライブラリに対して Clirr [2] という API の変更検出ツールを用いて調査を行い，セマンティックバージョニングが遵守されていないライブラリの存在を指摘している [3]．また，Raemaekers ら はソフトウェアが依存しているライブラリのアップデートへの追従が遅延することも指摘している．Kula ら [4] は，ライブラリのアップデートへの追従は追加のコストがかかるため優先されず，古いバージョンのライブラリに依存し続けるソフトウェアが存在することを指摘している．これは，ソフトウェア開発者がライブラリのバージョンを見ただけでは後方互換性の有無を正確に判断できず，アップデートの適用に伴う修正作業の必要性や影響範囲を見積もるのに追加のコストがかかるためであると著者らは考える．

また，セマンティックバージョニングを遵守している場合でも別の問題が挙げられる．後方互換性がない変更が行われた API が一つだけであったとしても，ライブラリのバージョンは後方互換性がないことを意味するものになる．そのため，具体的にどの API に後方互換性がないのかということについては別の手段により調べる必要がある．このような手間が発生してしまうこともアップデートへの追従が遅れる要因の一つになっていると著者らは考える．

このような問題を解決するために，本論文ではメソッド単位でバージョンを与える「メソッドレベルセマンティックバージョニング（Method-level Semantic Versioning）」という手法を提案する．また，実際に Java 言語のソースコードに対して提案手法を適用する

† 大阪大学大学院情報科学研究科，吹田市
Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565–0871 Japan
a) E-mail: j-hayasi@ist.osaka-u.ac.jp
DOI:10.14923/transinfj.2018JDP7074

ことによって，メソッドごとにバージョンを自動的に与えるツールを作成し，後方互換性のない変更が検出できることを確認する実験を行った．更に，Java ライブラリの後方互換性について調査した既存研究 [3] で用いられた API の変更を検出するツールである Clirr の出力結果と，著者らが作成したツールの出力結果を比較し，作成したツールでのみ検出できる変更を調査した．

その結果，Clirr では検出できなかったメソッドの属するパッケージ名の変更や，メソッドの引数リストの変更が検出できることが分かった．

2. セマンティックバージョニング

2.1 セマンティックバージョニングの定義

セマンティックバージョニングでは，一つのバージョンを「$X.Y.Z$」の形で表記する [1]．X, Y, Z は全て非負整数であり，それぞれメジャーバージョン，マイナーバージョン，パッチバージョンという．最初のリリースに与えるバージョンは 1.0.0 とし，以後のリリースに与えるバージョンは，現在のバージョンを基準に次のように決定する．

メジャーリリース　公開されている API に後方互換性のない変更が取り入れられたとき，メジャーバージョンを 1 増加させ，マイナーバージョンとパッチバージョンを 0 にする．

マイナーリリース　公開されている API に後方互換性のある変更が取り入れられたとき，あるいは新たに機能が公開されたとき，マイナーバージョンを 1 増加させ，パッチバージョンを 0 にする．メジャーバージョンは変更しない．

パッチリリース　誤った振る舞いを修正する後方互換性のある変更が取り入れられたときに限り，パッチバージョンを一つ増やす．メジャーバージョンとマイナーバージョンは変更しない．

メジャーバージョン，マイナーバージョン，パッチバージョンを増加させる変更を，それぞれメジャーレベル，マイナーレベル，パッチレベルの変更という．メジャーリリースにマイナーレベルやパッチレベルの変更を含むことや，マイナーリリースにパッチレベルの変更を含むことは許可されている．

文献 [1] ではこのほかにプレリリースの扱いなど，幾つかの仕様が定められているが，本論文では以上で定義されるバージョンのみを取り扱う．

2.2 セマンティックバージョニングは遵守されていない

Raemaekers らは，セマンティックバージョニングの原則が実際のプロジェクトにおいて遵守されているか調査した [3]．Raemaekers らは Maven Central Repository [5] に公開されている 22,205 個のソフトウェアを対象に調査した結果，本来後方互換性をもつべきであるマイナーリリースの 35.7%，パッチリリースの 23.8%に後方互換性のない変更が一つ以上含まれており，マイナーリリースやパッチリリースには平均で約 30 個の後方互換性のない変更が含まれていた．

2.3 バージョニングの問題点

バージョニングには幾つかの問題点がある．

まず，前節で述べたようにセマンティックバージョニングは遵守されていないことが少なくない．そのため，後方互換性があるはずのマイナーリリースやパッチリリースが行われたライブラリをアップデートした際に，実際には後方互換性がなかったために故障が発生してソースコードを修正しなければならなくなることが発生し得る．

次に，ライブラリをリリースする際にバージョニングは人の手で主観的に行われることが多い．例えば，Google Guava というライブラリではバージョニングのポリシーとしてセマンティックバージョニングを用いることが定められているが，ツールなどの指定や推薦はされていない [6]．これは，バージョニングを自動化したり補助したりといったツールが開発されていない，あるいはそれが普及していないことが理由としてあると考える．その一方で例えばコーディング規約やソフトウェアテストに関しては，コーディング規約が守られているか検証するツール（たとえば [7]）やテストを自動化するライブラリ（たとえば [8]）が存在する．

別の問題として，セマンティックバージョニングが遵守されていたとしても，図 1 のようにメジャーリ

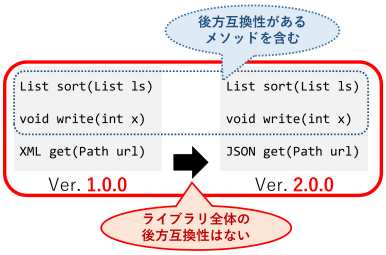

図 1　ライブラリ単位でのバージョニング

リースで行われた変更の中には後方互換性のあるもの（図 1 の例では sort(List) 及び write(int)）が含まれている可能性がある [3]．これはセマンティックバージョニングの定義で許可されていることだが，メジャーリリースされたライブラリを利用していても，そのリリースで変更された機能のうち後方互換性のなくなった機能は使用していない可能性がある．ライブラリの変更内容は，リリースノートなどが用意されていればそれを参照することで確認できるが，用意されていない場合や変更点が網羅しきれていない可能性が考えられる．これについても自動的に変更点を列挙することによって，リリースノートの網羅性を担保したり作成を補助したりすることにつながると考えられる．

3. メソッドレベルセマンティックバージョニング

本論文では，前章で述べた問題点を解消するために，ソースコードの変更履歴をもとにメソッド単位でセマンティックバージョニングを行い，各メソッドにバージョンを与える手法を提案する．以下，この手法を「メソッドレベルセマンティックバージョニング」と呼ぶ．

従来のバージョニングでは図 1 のようにライブラリ単位でしか後方互換性を示すことができないが，メソッドレベルセマンティックバージョニングを行うことによって，図 2 のように後方互換性がなくなったメソッド（図の例では get(Path)）を具体的に提示できるようになる．これにより，既にリリースされているソフトウェアに提案手法を適用した場合は，後方互換性のないメソッドが存在することを利用者に周知することが可能になる．また，これからリリースしようとしているソフトウェアに提案手法を適用した場合は，後方互換性のないメソッドが検出されたことをもってメジャーリリースとする判断材料に用いることができる．

本論文では，あるバージョン A のライブラリの利用者が，ソースコードを修正することなく新しいバージョン B のライブラリのメソッドを利用できるとき，そのメソッドにはライブラリのバージョン A とバージョン B の間で後方互換性があると定義する．

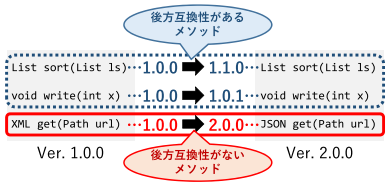

図 2　メソッド単位でのバージョニング

3.1 適用対象

提案手法は「メソッド」という名称のとおり，ある一連の手続きに名前を付けて他の場所から呼び出せる機能をもつ言語を対象としている．また，バージョン管理システムを用いてソースコードのバージョン管理が行われていることも提案手法を適用するための条件となる．提案手法は，主にライブラリの開発者が開発中のソフトウェアに対して適用し，バージョンを決定する一つの判断材料とされることを目的としているが，リリース済みのバージョンに対して適用することも可能である．

提案手法は様々なプログラミング言語に対して適用できると考えられるが，現時点では Java 言語に対してのみ提案手法を適用するツールを作成している．そのため，以降の説明も Java 言語を用いて行う．

3.2 アルゴリズム

各メソッドのバージョンを以下の手順で決定する．

（1） メソッドの追跡

（2） バージョンの決定

各手順について説明する．

(1) メソッドの追跡

あるリビジョン r_n で変更されたメソッドが，その前のリビジョン r_{n-1} におけるどのメソッドかを特定する必要がある．

リビジョン r_n で変更された各メソッドについて，前のリビジョン r_{n-1} に同じシグネチャ（メソッドの名称と引数の型 [9]）をもつメソッドが定義されていれば，それらは同じメソッドであると考えられる．一方，前のリビジョン r_{n-1} に同じシグネチャをもつメソッドがない場合，そのメソッドがリビジョン r_n で新たに追加されたメソッドであるのか，そのメソッドとは異なるシグネチャをもつメソッドのシグネチャを変更したものであるのかを判別しなければならない．

提案手法では，メソッドの本体の一致割合を基準にこれらを判別する．すなわち，リビジョン r_n のあるメソッド m について，メソッドの本体が一定割合以上一致しており，かつリビジョン r_n で削除されたメソッドがリビジョン r_{n-1} にある場合は，そのようなメソッドのうち最も一致している割合が大きいものをメソッド m の変更前のメソッドであるとし，そのようなメソッドがない場合はメソッド m がリビジョン r_n

で新たに追加されたメソッドであるとする．

(2) バージョンの決定

(1) で得られたメソッドの対応関係から，あるリビジョン r_n で変更されたメソッドは次の四つに分類できる．

追加メソッド　新たに追加されたメソッド
改名メソッド　シグネチャが異なるメソッド
修正メソッド　シグネチャが同じメソッド
削除メソッド　削除されたメソッド

この分類をもとに，リビジョン r_n において削除されたメソッドを除く各メソッドのバージョンを以下のように定める．

追加メソッド　1.0.0 とする．
改名メソッド　メソッドのシグネチャが変更された場合，あるいはメソッドのアクセス修飾子が狭まった場合（例えば，`public` 修飾子が `private` 修飾子に変更された場合など）はメジャーバージョンを一つ増やす．
修正メソッド　変更がバグ修正でなければマイナーバージョンを，バグ修正であればパッチバージョンを一つ増やす．

上記の修正メソッドの定義において，「変更がバグ修正である」ことは，その変更を含むコミットのコミットメッセージに “bug” または “fix” という単語が含まれていることと定義する．

4. Java メソッド自動バージョニングツール

バージョン管理システムの一つである Git でバージョン管理が行われている Java ソースコードに対して，メソッドレベルセマンティックバージョニングを自動的に適用するツールを Java 言語を用いて作成した．以下ではこれを「提案ツール」と呼ぶ．提案ツールでプログラムから Git を操作するためのライブラリとして JGit [10] を採用した．

4.1 ツールの概要

提案ツールの入力は Java ソースコードを含む Git リポジトリであり，出力はメソッドレベルセマンティックバージョニングの結果である．入力された Git リポジトリのソースコードにおいて定義されているメソッドのバージョンを計算する．**2.** で述べたセマンティックバージョニングの定義ではリリースごとにバージョニングを行うとされているが，提案ツールではコミットごとにセマンティックバージョニングを行う．

4.2 処理の流れ

提案ツールが入力リポジトリに対して行う処理の流れを次に示す．

Step 1：　コミット履歴を取得
Step 2：　メソッドの追跡
Step 3：　バージョンの決定

Step 2 と Step 3 は **3.2** で述べたアルゴリズムに対応する．以下，各 Step の具体的な処理について述べる．

Step 1：コミット履歴を取得

Git ではコミットを枝分かれさせてブランチを作ったり，複数のブランチを一つのコミットに統合（マージ）させたりでき，一般にコミットの履歴は図 3 のようなグラフになる．

提案ツールは，既定のブランチである `master` ブランチのコミットを解析し，枝分かれについては各コミットのファーストペアレントのみを解析する．ここで，`master` ブランチの最新コミットからファーストペアレントを辿り，古いコミットから順に並べたものを「(`master` ブランチの）コミット履歴」と呼ぶ．なお，コミットの「ファーストペアレント」とはそのコミットが行われた際に `HEAD` が指していたコミットである [11]．例えば，図 3 におけるコミット `2d3acf9` のファーストペアレントはコミット `5e3ee11` であり，コミット `5e3ee11` のファーストペアレントはコミット `420eac9`（コミット `420eac9` からブランチ `topic` をマージした場合）である．以下，コミット履歴において古い方から n 番目のコミットを「n 番目のコミット」と呼ぶ．

解析対象のブランチを `master` ブランチに限定したのは，解析した任意の 2 コミット間について，メソッドのバージョンを比較するだけで後方互換性の有無を判別できるようにするためである．

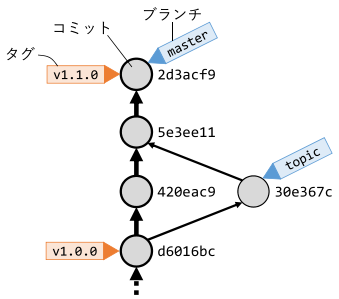

図 3　コミットグラフの例

Step 1 ではコミットを遡りながら直近のタグも取得する．タグとは Git においてコミットに付けられる別名で，例えば図 3 においてコミットを示す丸の左側に書かれた「`v1.0.0`」や「`v1.1.0`」である．

Step 2：メソッドの追跡

提案ツールでは，入力された Git リポジトリを Historage リポジトリ [12] に変換することでソースコードからメソッドの情報を抽出する．Historage はメソッドレベルでバージョン管理を行える Git ベースのバージョン管理システムであり，これを用いることで Git にメソッドの追跡を任せることができる．

提案ツールでは Kenja [12], [13] によって Git リポジトリを Historage リポジトリに変換する．Git リポジトリを Kenja に入力すると，ソースコードからメソッドの情報が抽出されて図 4 のような構造のディレクトリが構築される．

Web 上 [13] に公開されている Kenja が取得するメソッドの情報はメソッドの引数と本体（図 4 の `parameters` ファイルと `body` ファイル）のみであり，返り値の型やアクセス修飾子の情報は取得しない．メソッドの引数と本体だけではメソッドの後方互換性を判断するためには不十分であり，返り値の型及びアクセス修飾子の情報も考慮する必要があると考えられる．提案ツールでは Kenja が利用する Java 言語のパーサを返り値の型及びアクセス修飾子の情報（図 4 の `return` ファイル及び `modifiers` ファイル）も取得するように修正して利用する．

提案ツールでは，コミット履歴の n 番目のコミット c_n におけるメソッド f_n とその一つ前のコミット c_{n-1} におけるメソッド g_{n-1} について，`body` ファイルがバイト数を基準に 30%以上同じであるとき f_n と g_{n-1} が同じメソッドであるとし，メソッドのシグネチャが変更されたものとする．なお，基準値の 30%は Historage が採用している値と同じである [14]．また，Historage はリファクタリングなどによってメソッドが別のクラスに移動されたために元のクラスからは削除された場合も，移動元のメソッドと移動先のメソッドが上記の基準で同じメソッドである場合は追跡され，**3. 2** で述べた改名メソッドに分類される．同じでないと判断された場合は追跡されず，移動元のメソッドは削除メソッドに，移動先のメソッドは追加メソッドに分類される．

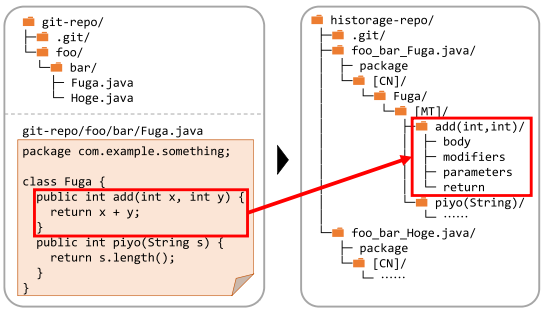

図 4　Git リポジトリ（左）を変換した Historage リポジトリ（右）

Step 3：バージョンの決定

Step 2 で求めたメソッドの対応関係から，**3. 2** の (2) で述べた方法に従ってバージョンを決定する．なお，提案ツールでは，メソッドの返り値の型が変更された場合あるいはアクセス修飾子が狭まった場合に後方互換性がない変更が行われたとする．

5. 実　　験

提案手法の評価を行うために，以下の二つの実験を行った．

実験 1　提案手法を用いて，本来後方互換性があるはずのマイナーリリース及びパッチリリースにおいて，後方互換性のない変更を含むリリースの割合を調査する．

実験 2　提案手法と既存の API 変更検出ツールである Clirr とを比較し，既存手法との差異を評価する．

なお，本実験では `public` クラスに含まれる `public` メソッドを API とみなして提案手法を適用した．

5. 1　実 験 対 象

Apache Commons プロジェクトにおいて開発されており，かつ以下の条件を全て満たす 28 ライブラリを対象に実験を行った．

- バージョンが Git リポジトリにおいてタグで示されている．
- バージョンがセマンティックバージョニングの形式（$X.Y.Z$）である．Z は省略されていても可とし，その場合は $Z = 0$ として取り扱った．
- バージョン 1.0.0 以上のリリースがある（ベータ版や Release Candidate 版を除く）．
- 複数回リリースされている（ベータ版や Release Candidate 版を除く）．

実験対象としたライブラリの名称とそのバージョンを表 1 に示す．

実験対象として Apache Commons プロジェクトを

表 1　実験対象のライブラリとそのバージョン．ライブラリ名は「Apache Commons」を省略して表記している．

ライブラリ名	バージョン
BCEL	5.0.0 (2001/12/15)～6.2.0 (2017/12/09)
BeanUtils	1.0.0 (2001/07/14)～1.9.3 (2016/09/26)
Chain	1.0.0 (2004/12/09)～1.2.0 (2008/06/02)
CLI	1.0.0 (2002/11/06)～1.4.0 (2017/03/09)
Codec	1.5.0 (2011/03/29)～1.11.0 (2017/10/20)
Collections	1.0.0 (2007/07/29)～4.1.0 (2015/11/28)
Compress	1.0.0 (2009/05/21)～1.13.0 (2016/12/29)
Configuration	1.0.0 (2004/10/11)～2.2.0 (2017/10/12)
CSV	1.0.0 (2014/08/14)～1.5.0 (2017/09/03)
DBCP	1.0.0 (2002/08/12)～2.2.0 (2017/12/27)
DbUtils	1.0.0 (2003/11/10)～1.7.0 (2017/07/20)
Digester	1.0.0 (2007/07/29)～3.2.0 (2011/12/13)
Email	1.0.0 (2005/09/27)～1.5.0 (2017/08/01)
Exec	1.0.0 (2009/03/15)～1.3.0 (2014/11/06)
FileUpload	1.1.0 (2005/12/24)～1.3.3 (2017/06/13)
IO	1.0.0 (2004/05/08)～2.5.0 (2016/04/22)
JCI	1.0.0 (2013/03/03)～1.1.0 (2013/10/14)
Jelly	1.0.0 (2005/06/12)～1.0.1 (2017/09/25)
JEXL	1.0.0 (2004/09/07)～3.1.0 (2017/04/14)
JXPath	1.0.0 (2007/07/29)～1.3.0 (2008/08/14)
Lang	1.0.0 (2002/10/04)～3.7.0 (2017/11/04)
Logging	1.0.3 (2003/04/07)～1.2.0 (2014/07/11)
Math	1.0.0 (2004/12/06)～3.6.1 (2016/03/21)
Net	1.0.0 (2003/02/23)～3.6.0 (2017/02/15)
Text	1.0.0 (2017/03/04)～1.2.0 (2017/12/12)
Validator	1.3.0 (2006/03/24)～1.6.0 (2017/02/21)
VFS	1.0.0 (2007/07/29)～2.2.0 (2017/10/06)
Weaver	1.0.0 (2014/03/16)～1.3.0 (2016/10/18)

選択したのは，プロジェクトの Web サイト [15] において，セマンティックバージョニングに相当するガイドラインが制定されているためである．なお，Apache Commons BCEL など一部のライブラリではパッチリリースが行われていないが，本実験は後方互換性の有無を調べるものであるため，パッチリリースの有無は本質でないと考える．そのため，パッチリリースのあるライブラリとないライブラリを区別なく取り扱うことによる実験結果への影響はない．

5.2　実験 1：誤ったバージョニングの検出

実験対象のライブラリに対して提案ツールを実行し，各リリースにおいて変更された `public` メソッドの後方互換性の有無を調査した．提案手法及び提案ツールではコミット単位でバージョニングを行っている．そのため本実験では，リリースのバージョンを示すタグが付けられたコミットと，そのバージョンのリリースを 1 対 1 に対応付け，リリース間のバージョンをメソッドごとに比較することでリリース間の後方互換性を判定した．

リリースの総数（# all）と後方互換性のない変更を含むリリースの個数（# incomp）を，メジャーリリース，マイナーリリース，パッチリリースに分けて集計したものを表 2 にまとめる．本来は後方互換性のあるはずであるマイナーリリースのうち 8 割以上，パッチリリースのうち 2 割以上には，実際には後方互換性のない変更が行われたメソッドが含まれていることが分かった．また，本来後方互換性のないはずであるメジャーリリースには全て後方互換性のない変更が一つ以上含まれており，マイナーリリースやパッチリリースで十分なリリースがメジャーリリースとされているような例はなかった．

表 2　リリースの種別と後方互換性のないリリースの割合

	# all	# incomp	割合
メジャーリリース	19	19	100.0%
マイナーリリース	163	131	80.4%
パッチリリース	65	17	26.2%

Apache Commons BCEL 5.1

```
public class ArrayType {
  public int hashcode() {
    return basic_type.hashCode() ^ dimensions;
  }
}
```

Apache Commons BCEL 5.2

```
public class ArrayType {
  public int hashCode() {
    return basic_type.hashCode() ^ dimensions;
  }
}
```

図 5　マイナーリリースに含まれていた後方互換性のない変更の例

Apache Commons IO 1.3.1

```
public class FileCleaner {
  public class Tracker {
    public boolean delete() {
      return deleteStrategy.deleteQuietly(new File(path));
    }
  }
}
```

Apache Commons IO 1.3.2

```
public class FileCleaningTracker {
  public boolean delete() {
    return deleteStrategy.deleteQuietly(new File(path));
  }
}
```

図 6　パッチリリースに含まれていた後方互換性のない変更の例

提案ツールが検出した後方互換性のない変更の例を図 5 及び図 6 に示す．図 5 はマイナーリリース，図 6 はパッチリリースに含まれていた後方互換性のない変更の例である．図 5 では，Apache Commons BCEL 5.1 から 5.2 へのマイナーリリースにおいて，ArrayType クラスの hashcode() メソッドが hashCode() に改名

されている．また，図 6 では，Apache Commons IO 1.3.1 から 1.3.2 へのパッチリリースにおいて，FileCleaner クラスのインナークラス Tracker の delete() メソッドが FileCleaningTracker クラスに移動されている．

このように，Apache Commons プロジェクトにはバージョニングのガイドラインが定められているにもかかわらず，後方互換性のない変更を含んでいる場合にマイナーリリースやパッチリリースとした場合があることが分かった．

後方互換性のあるはずであるマイナーリリースやパッチリリースに後方互換性のない変更が含まれている場合，そのライブラリの利用者がライブラリをアップデートした際に不具合が発生することが予想される．そこで，本実験で検出されたメソッドが実際に使用されている例を GitHub のキーワード検索を用いて調べた．その結果，検出されたどのメソッドもライブラリ外から使用しているソフトウェアは見つからなかった．この原因としては，検出されたメソッドが API として外部から利用されることを想定したメソッドではない，あるいは需要の少ないメソッドであったため，使用例がないと考えられる．

5.3 実験 2：既存手法との比較

実験 1 で提案ツールが検出した後方互換性のないリリースに対して Clirr を適用し，提案ツールの出力結果と Clirr の出力結果を目視によって比較した．

Clirr は，新旧二つのバージョンの JAR（Java Archive）ファイルを入力にとり，それらのバージョン間で後方互換性のない変更を以下の手順で検出するツールである．

（1） 新バージョンで削除されたメソッドと追加されたメソッドの全ての組み合わせに対して相違度を計算する．

（2） 最も相違度の小さいメソッドの組が対応するメソッドとし，引数の個数・引数の型・返り値の型のいずれかが異なっているかアクセス修飾子が狭まっている場合に後方互換性がないと判定する．

（3） (2) で処理したメソッドを除き，削除されたメソッドか追加されたメソッドがなくなるまで (1) から (3) を繰り返す．

上記の (1) における相違度は，メソッドが取る引数の個数に差がある場合は個数の差の絶対値を 1,000 倍した値であり，引数の個数に差がない場合は異なっている引数の型の数である [16]．例えば，メソッド funcA(int) と funcA(int, int, float) の相違度は $1{,}000 \times 2 = 2{,}000$ であり，funcA(int, float) と funcA(int, int) の相違度は 1 である．

実験の結果，次のような変更についての出力結果が提案ツールと Clirr で異なっていた．

a. メソッドの引数リストの変更
b. メソッドの属するパッケージ名の変更
c. 複数のメソッドの追加と削除

各変更の件数を表 3 に示す．それぞれについて，実際の検出例を説明する．

a. メソッドの引数リストの変更

Apache Commons IO 2.0 において，SizeFileComparator クラスの compare(Object, Object) メソッドの引数が変更されて compare(File, File) メソッドになったと提案ツールが出力した．一方，Clirr はこの変更に関して，compare(File, File) メソッドが追加されたという情報のみを出力しており，compare(Object, Object) メソッドが削除されたということは出力されていなかった．

b. メソッドの属するパッケージ名の変更

Apache Commons DBCP 2.0 において，org.apache.commons.dbcp パッケージが org.apache.commons.dbcp2 に改称された．このリリースに関して提案ツールは org.apache.commons.dbcp パッケージに含まれるメソッドが org.apache.commons.dbcp2 というパッケージに移動したという出力を行っていた．一方 Clirr の出力は，org.apache.commons.dbcp パッケージに含まれるクラスが削除されたという情報と，org.apache.commons.dbcp2 パッケージにクラスが追加されたという情報に分かれていた．

c. 複数のメソッドの追加と削除

Apache Commons IO 2.5 において，ByteArrayInputStream クラスに以下のような変更があった．

- toBufferedInputStream(InputStream, int) メソッドが追加
- toBufferedInputStream() メソッドが削除
- toInputStream() メソッドが追加

提案ツールはこれを以下のような変更として出力した．

表 3 提案ツールと Clirr で出力結果が異なっていた変更

変更内容	件数
a. メソッドの引数リストの変更	293
b. メソッドの属するパッケージ名の変更	65
c. 複数のメソッドの追加と削除	1

- toBufferedInputStream(InputStream, int) メソッドが追加
- toBufferedInputStream() メソッドが toInputStream() に改名

一方，Clirr は以下のように出力した．

- toBufferedInputStream() メソッドの引数が追加
- toInputStream() メソッドが追加

Clirr はメソッドの削除と追加，及び引数リストの変化などを検出でき，これらのパターンに基づいて変更内容を出力する．一方，提案ツールはメソッド本体の一致率が大きいものを同じメソッドであると判断し，シグネチャが変わっている場合は改名されたり引数リストが変更されたりしたものであるという出力を行う．実際，メソッド toBufferedInputStream() と toInputStream() の本体は，toBufferedInputStream(InputStream, int) よりも一致割合が大きかった．そのため，toBufferedInputStream() と toInputStream() が同じメソッドであり，前者の名称が後者に変更されたという出力になったと考えられる．

この結果から，提案ツールはメソッド名の変更を検出することができるといえる．これにより，ライブラリのバージョンアップによってメソッド名が変更された場合に，ライブラリの利用者が提案ツールの出力結果を利用して改名されたメソッドの呼び出し元を一括置換することができるようになる．これはライブラリアップデートへの対応を省力化できると考えられ，その点において提案ツールは Clirr よりも優れていると考える．

6. 妥当性への脅威

6.1 実験対象の選択

本研究では実験 2 を実験 1 に続いて行ったため，どちらも Apache Commons プロジェクトのライブラリを用いて実験した．Apache Commons プロジェクトを実験対象としたのは，実験 1 においてセマンティックバージョニングを採用しているという前提が必要であったためである．しかし，実験 2 についてはセマンティックバージョニングの制約は必要でない．実験 2 を他のライブラリで実験した場合に異なる実験結果が得られる可能性がある．

6.2 public メソッドと API の関係

本実験では，`public` クラスに含まれる `public` メソッドを全て API とみなした．Java 言語では，`public` クラスに含まれる `public` メソッドは全てライブラリ外部から呼び出すことができ，API を開発者が指定することができない．そのため，API が含まれるクラスだけでなく，あるパッケージに定義されているクラスを別のパッケージから参照したい場合も，そのクラスを `public` とする必要がある．したがって，そのクラスが内部処理のためだけのものであったとしても，それに定義されている `public` メソッドは本実験の対象に含まれている．すなわち，本実験で API としたメソッドには，開発者が API とは想定していないメソッドが含まれている可能性がある．それらのメソッドを特定して実験対象から除いた場合，実験結果が変化する可能性がある．

6.3 メソッドの後方互換性

提案ツールでは，ライブラリの利用者がライブラリのバージョンを上げた際にソースコードの修正を行うことなく同じメソッドを使い続けられる場合に，そのメソッドに後方互換性があるとしてバージョニングを行っている．しかし，メソッドのシグネチャが同じでも，メソッドの中身が変更されたことでそのメソッドの入力に対する出力が変化した場合も，後方互換性がないと考えられる．提案ツールではメソッドのシグネチャの変化のみをもって後方互換性の有無を判断しているため，メソッドの入出力の変化に関しては考慮していない．より正確に後方互換性の有無を判別するためには，メソッドの入出力が変化していないかどうかを解析する必要がある．

また，あるコミット c_1 で後方互換性のない変更が行われ，その後に別のコミット c_2 で c_1 を適用する前（c_0 とする）と後方互換性のある状態に戻される場合が考えられる．しかしながら，提案手法及び提案ツールではコミットごとにメソッドの後方互換性を判別しバージョニングを行っているため，このような場合は c_1，c_2 のそれぞれでメジャーバージョンが一つ増える．したがって，このメソッドの c_0-c_2 間における後方互換性をバージョンから判断すると誤りとなる．本研究では，このような場合を考慮するとバージョンの決定が困難となるため，上述のような扱いになっている．

6.4 「バグ修正コミット」の定義

提案ツールでは，後方互換性のある変更をマイナーレベルとパッチレベルに分類するために，変更が行われたコミットのコミットメッセージに特定の文字列が含まれているかどうかによって，その変更がバグ修正のためのものであるかどうかを判断した．本研究では

後方互換性の有無を議論の焦点としたため，また，正確にバグ修正であるかどうかを判断することは困難であるとの考えからこのような実装とした．しかし，バグ修正が行われたコミットのコミットメッセージに特定の単語が含まれていることを前提とすることの是非については議論の余地があると考えられる．

7. むすび

本研究では，メソッドレベルセマンティックバージョニングを提案し，それを Java 言語のメソッドに適用するツールを作成した．また，作成したツールを Apache Commons プロジェクトの 28 ライブラリに対して使用することで，後方互換性のあるはずであるリリースに含まれていた後方互換性のない変更を検出することができた．更に，既存ツールである Clirr の出力結果と比較を行うことにより，Clirr が検出できない変更を提案ツールを用いることで検出できることが分かった．

提案ツールはシグネチャや返り値の型が変更された場合に後方互換性がないと判断してバージョニングを行っている．しかし，より正確には，型（クラス）の継承関係やメソッドの返り値自体の変化，更には副作用なども考慮する必要があると考えられ，これらを考慮することによって後方互換性の有無をより正確に判定できるようにしたいと考えている．

本研究では，提案ツールと Clirr の比較を行った．しかし，API の変更を検出することができるツールは他にも存在し，例えば Moreno らが開発した ARENA [17] というツールがある．ARENA は検出した API の変更をリリースノートとして出力する Web アプリケーションである．ARENA はメソッド名の変更を追跡することが可能であり，提案ツールとメソッドの追跡精度や出力に要する時間などを比較したいと考えている．

謝辞　本研究は，科学研究費補助金基盤研究（B）（課題番号：17H01725）の助成を得て行われた．

文　献

[1] T. Preston-Werner, “Semantic versioning 2.0.0,” June 2013. https://semver.org/spec/v2.0.0.html
[2] L. Kühne, “Clirr,” Sept. 2005. http://clirr.sourceforge.net
[3] S. Raemaekers, A. van Deursen, and J. Visser, “Semantic versioning versus breaking changes: A study of the maven repository,” 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp.215–224, Sept. 2014.
[4] R.G. Kula, D.M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?,” Empirical Software Engineering, vol.23, no.1, pp.384–417, Feb. 2018.
[5] MvnRepository, “Maven repository,” Jan. 2018. https://mvnrepository.com/
[6] C. Decker, “Releasepolicy - google/guava wiki,” Google, Sept. 2017. https://github.com/google/guava/wiki/ReleasePolicy
[7] Checkstyle, “Checkstyle,” Oct. 2018. http://checkstyle.sourceforge.net
[8] JUnit Team, “JUnit,” Sept. 2018. https://junit.org/
[9] J. Gosling, B. Joy, G.L. Steele, G. Bracha, and A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st ed., Addison-Wesley Professional, 2014.
[10] Eclipse Foundation, “JGit,” Oct. 2018. http://www.eclipse.org/jgit/
[11] S. Chacon and B. Straub, Pro git, Apress, 2014.
[12] K. Fujiwara, H. Hata, E. Makihara, Y. Fujihara, N. Nakayama, H. Iida, and K. Matsumoto, “Kataribe: A hosting service of historage repositories,” Proc. 11th Working Conference on Mining Software Repositories, pp.380–383, New York, USA, 2014.
[13] K. Fujiwara, “niyaton/kenja: Kenja repository,” Dec. 2017. https://github.com/niyaton/kenja
[14] H. Hata, O. Mizuno, and T. Kikuno, “Historage: Fine-grained version control system for java,” Proc. 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution, pp.96–100, IWPSE-EVOL ’11, ACM, New York, NY, USA, 2011.
[15] Apache Software Foundation, “Apache Commons – Versioning Guidelines,” Oct. 2018. https://commons.apache.org/releases/versioning.html
[16] L. Kühne, “clirr/MethodSetCheck.java at master - ebourg/clirr,” Sept. 2005. https://github.com/ebourg/clirr/blob/master/core/src/java/net/sf/clirr/core/internal/checks/MethodSetCheck.java
[17] L. Moreno, G. Bavota, M.D. Penta, R. Oliveto, A. Marcus, and G. Canfora, “Arena: An approach for the automated generation of release notes,” IEEE Trans. Softw. Eng., vol.43, no.2, pp.106–127, Feb. 2017.
[18] 藤原賢二，吉田則裕，飯田　元，“構文情報リポジトリを用いた細粒度リファクタリング検出手法，” 情処学論，vol.56, no.12, pp.2346–2357, Dec. 2015.

（2018 年 11 月 27 日受付，2019 年 4 月 24 日再受付，
7 月 12 日早期公開）

林　　純一

平成 30 年大阪大学基礎工学部情報科学科卒業．同年より同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博士前期課程在学中．

肥後　芳樹　（正員）

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同大学院情報科学研究科コンピュータサイエンス専攻助教．2015 年同准教授．博士（情報科学）．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．日本ソフトウェア科学会，IEEE 各会員．

楠本　真二　（正員）

昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価，プロジェクト管理に関する研究に従事．IPSJ，IEICE，JSSST，IEEE，JFPUG，PM 学会，SEA 各会員．