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.

メソッドレベルセマンティックバージョニングの提案

林　　純一†　　肥後　芳樹†　　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: †{j-hayasi,higo,kusumoto}@ist.osaka-u.ac.jp

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，機能追加や変更，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の 1 つに，セマンティックバージョニングと呼ばれる手法がある．この方法ではライブラリ全体に 1 つのバージョンを付与するため，それだけでは後方互換性がない変更を特定できない．また，先行研究はセマンティックバージョニングの原則に従っていないライブラリがあると指摘している．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 言語のメソッドにバージョニングを行うツールを作成した．さらに，提案手法および作成したツールの有用性を示すために，提案手法を用いて得たメソッドのバージョンからセマンティックバージョニングに従っていないリリースとその原因となっているメソッドを検出する実験を行った．実験の結果，メソッドに与えたバージョンをもとに後方互換性が失われたメソッドを特定し，誤ったバージョニングが行われたリリースを検出できた．

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

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や不具合の修正が行われる．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．バージョンの表現やそれを決定する方法はライブラリによって様々であるが，その 1 つとしてセマンティックバージョニング (Semantic Versioning) という，1 つのバージョンを 3 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．しかし，先行研究 [2] は既存の Java 言語製ライブラリに対して Clirr [3] 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．すなわち，後方互換性の有無がバージョンに反映されていない可能性がある．さらに，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時間を要す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考えられる．後方互換性がない変更が行われた API が 1 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 1 つになっていると考えられる．

このような問題を解決するために，本研究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．さらに，Java 言語のソースコードに対して提案手法を実現するツールを作成し，これらの有用性を確認する実験を行った．

先行研究 [2] の調査で用いられた Clirr は入力が JAR (Java Archive) ファイルであるため，メソッドの対応付けと後方互換性の判定は JAR ファイルを生成するリリース単位で行われる．一方，本研究で作成したツールはソースコードが管理されている Git リポジトリを入力とし，コミット単位でこれらの処理を行う．そのため Clirr よりも細かい間隔でメソッドの対応関係を調べられ，より正確にリリース間のメソッドの対応関係を求められると考えられる．

本稿の構成は以下のとおりである．2 章ではセマンティックバージョニングの定義と問題点について述べる．3 章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する．4 章では提案手法を Java 言語のメソッドに適用するツールについて述べる．5 章では提案手法およびツールの有用性を評価する実験について述べる．6 章では実験結果の考察を行う．7 章では妥当性の脅威について述べる．最後に 8 章で本研究をまとめ，今後の課題について述べる．

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 [1] では，1 つのバージョンを「$X.Y.Z$」の形で表記する．X, Y, Z はすべて非負整数であり，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．

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 1 増加させる．

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．メジャーバージョン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．

バージョンの比較は，メジャーバージョン，マイナーバージョン，パッチバージョンの順に見て，最初に異なる部分の数値を比較することで行う．例えば 1.0.0 $<$ 2.0.0 $<$ 2.1.0 $<$ 2.1.1 のようになる．

[1] ではこの他にプレリリースの扱いなど，いくつかの仕様が定められているが，本研究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

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研究 [2] がある．文献 [2] は，本来後方互換性を持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 1 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るという調査結果を報告している．また，セマンティックバージョニングは徐々に守られるようになってきているが，その速さは緩やかであると報告している．

2.3 バージョニングの問題点

バージョニングにはいくつかの問題点があると考える．

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エラーや不具合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．

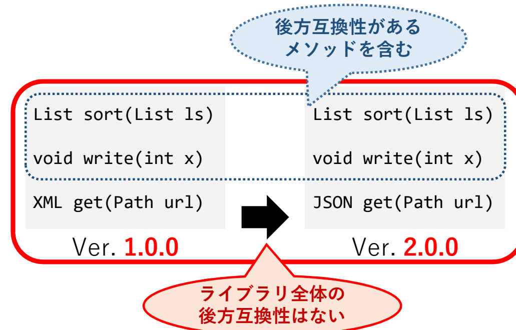

図 1 ライブラリ単位でのバージョニング

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 [4] が，ツールなどの指定や推薦はされていない．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（例えば [5]）やテストを自動化するライブラリ（例えば [6]）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)` および `write(int)`）が含まれている可能性がある [2]．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

本研究では，前章で述べた問題点を解消するために，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．

3.1 前　　提

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

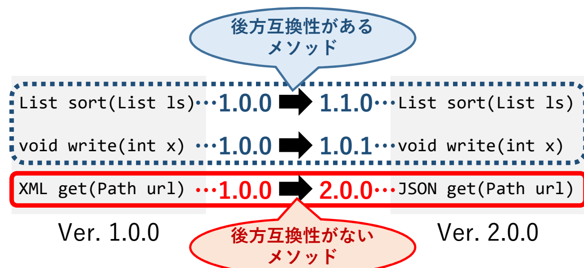

図 2　メソッド単位でのバージョニング

きに名前を付けて他の場所から呼び出せる機能を持つ言語を前提とする．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を前提とする．

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

一般に，1 つのソースコードファイルには複数のメソッドの定義が含まれる．そのため，ソースコードからそれぞれのメソッドの情報を抽出し，変更前後のソースコードにおけるメソッドの対応関係を調べる必要がある．提案手法では，ソースコードからメソッドを抽出し，2 つのソースコード間における対応関係を以下のように割り出す．

次に，ソースコード A と B において対応するメソッドの組を求める．ソースコード A と B でメソッドのシグネチャが同じものは対応すると考えるのが自然であろう．一方，メソッド名が変更された場合など，シグネチャは異なるが「同じ」メソッドであるとすべき場合も考えられる．これについては，A と B におけるメソッドのボディの類似度を基準にメソッドが対応しているかどうかを判断する．

(2) バージョンの決定

メソッドの対応関係から，ソースコード A から B にかけて変更されたメソッドを以下の 4 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，ソースコード B における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，あるいはメソッドを呼び出せる範囲が狭まった場合はメジャーバージョンを 1 増加させる．そうでない場合で，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 1 増加させる．

修正メソッド　変更に後方互換性がない場合はメジャーバージョンを 1 増加させる．後方互換性がある場合は，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 1 増加させる．

削除メソッド　処理の都合上メジャーバージョンを 1 増加させる．

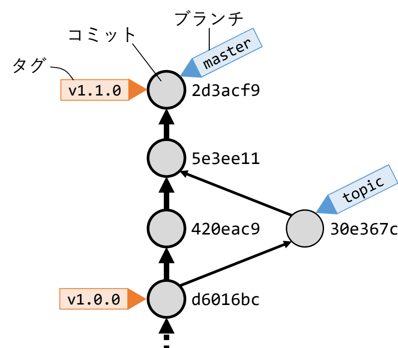

図 3　コミットグラフの例

4.　Java メソッド自動バージョニングツール

本研究では，バージョン管理システムの 1 つである Git でバージョン管理が行われている Java 言語の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 [8] を採用した．

4.1　ツールの概要

提案ツールの入力は Java 言語の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 章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毎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毎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 節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 (branch) を作ったり，複数のブランチを 1 つのコミットに統合（マージ，merge）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．そのため，どのブランチを解析するか，枝分かれした部分をどのように解析するかという問題が生じる．

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とにした．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

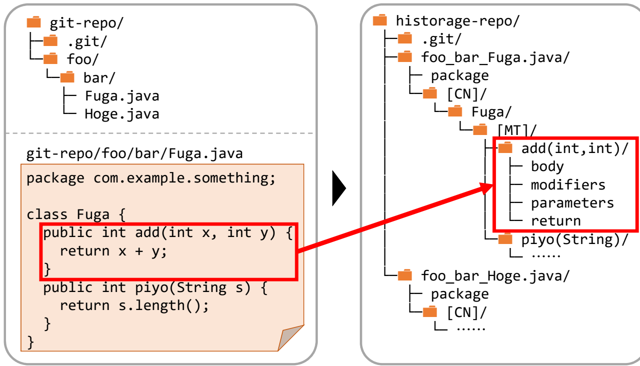

図 4　Git リポジトリ（左）を変換した Historage リポジトリ（右）

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 [9]．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．

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 [10] 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る．提案ツールでは Kenja [10] [11] 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．

[11] で公開されている Kenja が取得するメソッドの情報はメソッドの引数とボディ（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．メソッドの引数とボディ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，返り値の型およ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 [12] を返り値の型およびアクセス修飾子の情報（図 4 の `return` ファイルおよび `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 1 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 60% 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，60% という数字は JGit の設定の既定値である．

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 節の (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

提案手法および提案ツールの有用性を示すために，バージョン管理システム Git で管理されている Java 言語で書かれたライブラリがセマンティックバージョニングに基づいていると仮定して，それに従っていないリリースを検出する実験を行った．すなわち，本来後方互換性があるはずのマイナーリリースおよびパッチリリースにおいて，後方互換性のない変更が行われた `public` メソッドを検出する実験を行った．以下では，本実験で検出するリリースおよびメソッドを「セマンティックバージョニングに違反した」リリースおよびメソッドと呼ぶ．

5.1 実験対象

本実験では，先行研究 [2] において研究対象に用いられている Maven Repository [13] にあるライブラリのうち利用数の多い表 1 に示す 9 つのライブラリを実験対象とした．より多くのソフトウェアで利用されているライブラリにおけるセマンティックバージョニングの違反の影響が大きいと考えられるため，このような選定基準とした．

表 1 の項目名で用いた表現とその意味は以下のとおりである．

対象範囲　開発者がタグとして設定したバージョンのうち，本実験で対象とした範囲

リリース数　対象範囲で行われたリリースの回数

メソッド数　`public` メソッドの数

ライブラリ名を省略する際は，表 1 でライブラリ名の左に記載した L1–L9 を用いる．

実験対象のライブラリのうち Google Guava に関しては，セマンティックバージョニングを用いてバージョニングを行うことがリリースポリシー [4] に明言されている．また，Apache Commons プロジェクトのライブラリ (L1–L3) に関しては，セマンティックバージョニングに相当すると考えられるガイドライン [14] が制定されていることを確認した．

表 1 に示したように一部のリリースにはタグが設定されて

表 1　実験対象のライブラリと規模

	ライブラリ名	対象範囲	リリース数	メソッド数
L1	Apache Commons IO	1.0.0– 2.5.0	14	3,183
L2	Apache Commons Lang	1.0.0– 3.7.0	19	13,150
L3	Apache Log4j 2	2.0.0–2.10.0	17	16,508
L4	Google Guava	1.0.0–23.6.0	29	34,287
L5	Jackson Databind	2.0.0– 2.9.1	28	14,359
L6	Java Servlet API	3.0.1– 4.0.0	3	190
L7	JUnit4	3.8.2–4.12.0	10	5,567
L8	Logback	1.0.0– 1.2.2	23	2,408
L9	Mockito	1.0.0– 2.1.0	35	10,785

いなかったが，タグが設定されていたリリースを対象に実験を行った．また，ベータ版などのプレリリースやメジャーバージョンが 0 であるリリースは対象としなかった．

5.2 実験方法

対象のライブラリに対して，以下の手順で実験を行った．

Step 1 提案ツールを実行し，各メソッドの後方互換性を調査

Step 2 ライブラリ全体の後方互換性を決定

Step 3 実際のライブラリのバージョン変化と比較

ここでは Step 2 と Step 3 について詳しく述べる．

Step 2: ライブラリ全体の後方互換性を判定

開発者が与えたライブラリのバージョンから判断される後方互換性と比較するために，リリースに 1 つでも後方互換性が失われたメソッドを含むとき，そのライブラリ全体の後方互換性が失われたと判定した．

実験対象のライブラリのうち Java Servlet API を除くライブラリではソフトウェアテストが実施されており，JUnit [6] を利用したソースコードが含まれていた．JUnit ではテストケースを `public` メソッド（テストメソッド）として定義するため，単に `public` なメソッドを収集するとテストメソッドが含まれてしまう．テストメソッドはあくまでソフトウェアテストに用いられるコードであり，実際のソフトウェア開発においてライブラリのテストコードを利用することはほとんどないと考えられる．したがってライブラリ全体の変更のレベルを求める際にテストメソッドの後方互換性を考慮する必要はないと判断し，以下のメソッド呼び出しを含むメソッドを除外してライブラリ全体の後方互換性を判定した．

`assertArrayEquals`, `assertEquals`, `assertFalse`, `assertNotNull`, `assertNotSame`, `assertNull`, `assertSame`, `assertThat`, `assertTrue`.

Step 3: 実際のライブラリのバージョン変化と比較

本来はすべての変更に後方互換性があるはずのマイナーリリースおよびパッチリリースについて，実際には後方互換性のない変更が `public` メソッドに行われたリリースとその原因となったメソッドを検出した．

5.3 実験結果

まず，セマンティックバージョニングに違反していると判断されたリリースの数と総リリース数に対する割合を表 2 に示す．R_{all} は表 1 に示した区間の最初のバージョンを除いたリリース数，R_{incor} はセマンティックバージョニングに違反したリリースの数である．実験対象のすべてのライブラリにおいてセマンティックバージョニングに違反したリリースが存在し，その割合は Apache Log4j 2 や Jackson Databind で多く，Google Guava で少ない結果となった．

次に，後方互換性のないメソッドの変更がどのレベルのリリースで行われているかを調べた．その結果を表 3 にまとめる．M_{major} は後方互換性のない変更が行われたことがある `public` メソッド数，M_{incor} は後方互換性のない変更がマイナーリリースあるいはパッチリリースで行われたことがある `public` メソッド数である．これを棒グラフにしたものを図 5 に示す．各項目の上の棒（橙色）は後方互換性のないメソッドの変更が本来後方互換性のあるマイナーリリースあるいはパッチリリースで行われたことのあるメソッドの数，下の棒（青色）は後方互換性のない変更がすべて正しくメジャーリリースにおいて行われたメソッドの数を示す．

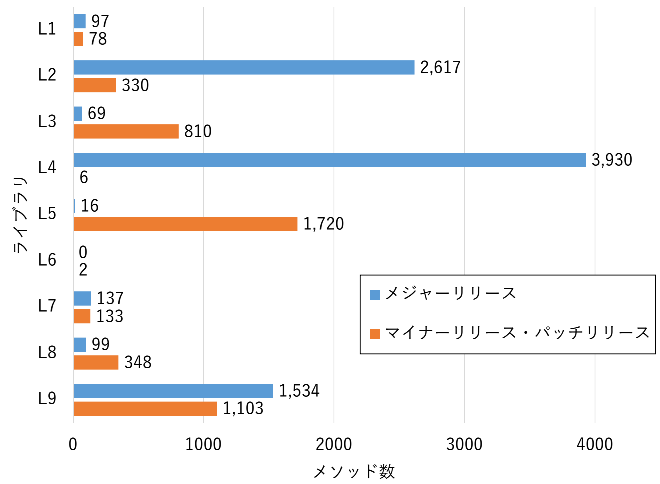

図 5　後方互換性のない変更が行われたメソッド

Google Guava (L4) は後方互換性のない変更が行われたメソッドが最も多いが，マイナーリリースやパッチリリースにおいて行われたメソッドは最も少ないライブラリであることが分かった．一方で Apache Commons Log4j 2 (L3) や Jackson Databind (L5) は後方互換性のないほとんどの変更が本来後方互換性のあるリリースで行っていることが分かった．

6. 考察

表 3 の結果から，1 リリース当たり平均 27.0 個のメソッドが

表 2　セマンティックバージョニングに違反したリリースの数

ライブラリ	R_{all}	R_{incor}	割合
Apache Commons IO	13	7	53.8%
Apache Commons Lang	17	8	47.1%
Apache Log4j 2	16	16	100.0%
Google Guava	28	2	7.1%
Jackson Databind	27	20	74.1%
Java Servlet API	2	1	50.0%
JUnit4	9	6	66.7%
Logback	22	17	77.2%
Mockito	34	22	64.7%

表 3　後方互換性のない変更が行われたことがあるメソッド

ライブラリ	M_{major}	M_{incor}	割合
Apache Commons IO	175	78	44.6%
Apache Commons Lang	2947	330	11.2%
Apache Log4j 2	879	810	92.2%
Google Guava	3936	6	0.2%
Jackson Databind	1736	1720	99.1%
Java Servlet API	2	2	100.0%
JUnit4	270	133	49.3%
Logback	447	348	77.9%
Mockito	2637	1103	41.8%

マイナーリリースやパッチリリースにおいて行われていることが分かった．これは先行研究 [2] で Clirr を用いて行われた実験結果と概ね一致する．ただし，提案ツールは Clirr よりも細かい間隔でメソッドの対応関係を調べているため，メソッドの追跡精度が向上した可能性が考えられる．そのため，この違いを評価するために本実験の結果を精査し，追加の実験を行う必要があると考える．

7. 妥当性への脅威

7.1 メソッドの後方互換性

提案ツールで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，さらには副作用なども考慮する必要があると考えられる．これらを考慮した場合に実験結果が変わる可能性がある．

7.2 「バグ修正コミット」の定義

提案ツールでは，変更のレベルのうち後方互換性のあるマイナーレベルとパッチレベルの分類に変更が行われたコミットのコミットメッセージをもとにその変更がバグ修正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是非について議論の余地があると考えられる．

7.3 他のプログラミング言語におけるメソッドレベルセマンティックバージョニング

本研究で行ったメソッドレベルセマンティックバージョニングの定義は，メソッドに相当する機能を備えていれば適用可能であるが，それを適用するツールおよびそれを用いた実験は Java 言語のみを対象に行った．そのため，他のプログラミング言語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適用した場合に実験結果が変わる可能性がある．

8. あ と が き

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，作成したツールを実際のライブラリに対して用いてメソッドレベルセマンティックバージョニングを行い，誤ったバージョニングが行われているリリースとその原因となったメソッドを検出する実験を行った．それにより，先行研究 [2] が示した結果と同程度に検出結果が得られたため，提案手法に一定の有用性があると考えられる．

提案ツールはコミット単位でメソッドの追跡と後方互換性の判定を行うため，これらをリリース単位で行う Clirr よりも正確にメソッドの追跡が行えることが期待されるが，本研究ではこの点についての評価実験が行えていない．

これを踏まえて以下の 3 点を今後の課題として挙げる．

- 提案ツールおよび Clirr によるメソッドの追跡精度を比較する評価実験
- メソッドの後方互換性の判断において，クラスの継承関係やメソッドの返り値自体の変化および副作用についても考慮
- メソッドレベルセマンティックバージョニングの結果をソフトウェア開発者にどのようにフィードバックするか検討

謝辞 本研究は，日本学術振興会科学研究費補助金基盤研究（S）（課題番号：JP25220003）の助成を得て行われた．

文　献

[1] “Semantic Versioning 2.0.0”. https://semver.org/spec/v2.0.0.html
[2] S. Raemaekers, A. Van Deursen, and J. Visser, “Semantic versioning versus breaking changes: A study of the maven repository,” Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference onIEEE, pp.215–224 2014.
[3] “Clirr”. http://clirr.sourceforge.net/index.html
[4] “ReleasePolicy · google/guava Wiki”. https://github.com/google/guava/wiki/ReleasePolicy
[5] “Checkstyle”. http://checkstyle.sourceforge.net/
[6] “JUnit”. https://junit.org/
[7] “Defining Methods (The Java™ Tutorials > Learning the Java Language > Classes and Objects)”. https://docs.oracle.com/javase/tutorial/java/javaOO/methods.html
[8] “JGit”. http://www.eclipse.org/jgit/
[9] S. Chacon and B. Straub, Pro git, Apress, 2014.
[10] K. Fujiwara, H. Hata, E. Makihara, Y. Fujihara, N. Nakayama, H. Iida, and K. Matsumoto, “Kataribe: A hosting service of historage repositories,” Proceedings of the 11th Working Conference on Mining Software RepositoriesACM, pp.380–383 2014.
[11] “Kenja”. https://github.com/niyaton/kenja
[12] “kenja-java-parser”. https://github.com/niyaton/kenja-java-parser
[13] “Maven Repository”. https://mvnrepository.com/
[14] “Apache Commons – Versioning Guidelines”. https://commons.apache.org/releases/versioning.html