特別研究報告

題目

メソッドに対するセマンティックバージョニング手法の提案

指導教員

楠本 真二 教授

報告者

林 純一

平成 30 年 2 月 14 日

大阪大学 基礎工学部 情報科学科

平成 29 年度 特別研究報告

メソッドに対するセマンティックバージョニング手法の提案

林 純一

内容梗概

ソフトウェア開発において利用されるライブラリは，機能追加や変更，バグ修正などが行われることで変化する．ライブラリが何度変化したものであるかを区別するために，バージョンという数値が用いられる．バージョンを付与することをバージョニングといい，その手法の 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．この手法は，バージョンから後方互換性の有無を判断できるようにバージョニングを行う方法である．バージョンはライブラリを区別するために付与されるものであるため，1 つのライブラリに対して 1 つ与えられる．そのため，バージョンからライブラリに後方互換性がないという判断はできるが，ライブラリが提供するすべての機能に後方互換性がないのか，あるいは一部の機能だけ後方後方互換性がないのか，ということまでは判断できない．すなわち，ライブラリのバージョンだけでは特定の機能について後方互換性の有無を判断できないという問題がある．また，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する先行研究がある．本研究ではこれらの問題点を解決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う手法を提案し，Java 言語のメソッドに対して提案手法を適用するツールを作成した．さらに，9 つのライブラリについてセマンティックバージョニングの原則に従っていないリリースを調査し，その原因となっているメソッドを特定する実験を行った．実験の結果，9 つすべてのライブラリにセマンティックバージョニングに反したリリースがあり，それらのリリースにおいて 1 リリースあたり平均約 55 個のメソッドに後方互換性のない変更が行われていることが明らかになった．

主な用語

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，リポジトリマイニング

目次

1 まえがき 1

2 準備 3

2.1 セマンティックバージョニングの定義 3

2.2 既存のバージョニング手法における問題点 4

2.3 リポジトリマイニング 5

3 提案手法 6

3.1 前提 6

3.2 アルゴリズム 6

4 実装 9

4.1 ツールの概要 9

4.2 処理の流れ 9

5 実験 13

5.1 実験対象 13

5.2 実験方法 13

5.3 実験結果 15

6 考察 19

6.1 実験結果の考察 19

6.2 先行研究との差異 19

7 妥当性への脅威 20

7.1 メソッドの後方互換性 20

7.2 「バグ修正コミット」の定義 20

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 20

8 あとがき 21

謝辞 22

参考文献 23

図目次

1 ライブラリ単位でのバージョニング 4
2 メソッド単位でのバージョニング 6
3 メソッドの対応関係の例 8
4 提案ツールのシステム概要 9
5 コミットグラフの例 10
6 Git リポジトリとそれを変換した Historage リポジトリ 11
7 セマンティックバージョニングの原則に違反したリリースの割合 15
8 本来後方互換性があるはずのリリースに含まれている後方互換性のないメソッドの数 . 17
9 Apache Commons Lang において検出された後方互換性のない変更の例 18

表目次

1 実験対象のライブラリと規模 14

2 セマンティックバージョニングの原則に違反したリリース 16

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 16

1　まえがき

ソフトウェア開発は，一般にライブラリと呼ばれる複数の機能が実装されたプログラムを利用して行われる．このライブラリは不変なものではなく，機能の追加や変更あるいはバグの修正が行われて変化する．ライブラリが何度変化したものであるかを区別するために，バージョンという数値が用いられる．バージョンを付与することをバージョニングといい，その手法の 1 つにセマンティックバージョニング (Semantic Versioning) と呼ばれる手法が提唱されている [13].

セマンティックバージョニングでは，ライブラリにおいて公開されている API (Application Programming Interface) 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，セマンティックバージョニングの原則が遵守されていないライブラリが存在していることが Raemaekers らの調査 [22] で明らかにされている．一方，セマンティックバージョニングに従っている場合でも，バージョンからライブラリのいずれかの API に後方互換性のない変更があるかどうかを知ることができるが，具体的に後方互換性のない変更が行われた API を特定するには，リリースノートなどの別の手段を用いて調べなければならない.

Bauml らは OSGi (Open Service Gateway initiative) フレームワーク [11] のバンドルに対して自動的にバージョンを与える手法を提案している [20]．Bauml らの手法は OSGi のコンポーネントを対象としているため，OSGi フレームワークを利用していないライブラリには適用できない.

本研究では，メソッド単位でバージョンを付与する「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．本手法によって各メソッドにバージョンを付与することで，メソッドごとの後方互換性をバージョンから判別できるようになる．これにより特定のメソッドに対して後方互換性があるかどうかを容易に調べられるようになると考えられる．また，各メソッドの後方互換性からライブラリ全体の後方互換性を判断することもできるようになり，従来の手作業で行われているバージョニングの誤りが低減できると考えられる.

さらに本研究では，Java 言語のソースコードに対して提案手法を適用するツールを作成した.

また，9 つのライブラリについてセマンティックバージョニングの原則に反したリリースを調べ，その原因となっているメソッドを特定する実験を行った．実験の結果，9 つすべてのライブラリにセマンティックバージョニングの原則に反したリリースが存在し，それらのリリースにおいて 1 リリースあたり平均約 55 個のメソッドに後方互換性のない変更が加えられていたことが分かった.

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従来のバージョニング手法に存在する問題点，およびリポジトリマイニングについて述べる．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　準備

本章では，セマンティックバージョニングの定義と既存のバージョニング手法における問題点，およびリポジトリマイニングについて述べる．

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，1 つのバージョンを「$X.Y.Z$」の形で表記する [13]．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．なお，「より大きい」方がより新しいバージョンであることを意味する．

開発初期においてメジャーバージョンを 0 として「$0.Y.Z$」の形のバージョンを与えることができる．メジャーバージョンが 0 である間は，先述の基準に依らずマイナーバージョンやパッチバージョンを上げることができる．また，このようなバージョンにおいて公開されている API は仕様などがまだ安定していないと考えるべきである．

プレリリース版を表すために，「$X.Y.Z$-alpha.1」のように，パッチバージョンの後にハイフンとドット区切りの識別子を続けることができる．識別子には ASCII の英数字とハイフンが利用できる．あるバージョンのプレリリース版は，通常の（プレリリース版を表す文字列のない）バージョンよりも小さいとする．例えば $1.0.0\text{-alpha.1} < 1.0.0$ である．なお，同じバージョンのプレリリース版同士は辞書順で比較する．

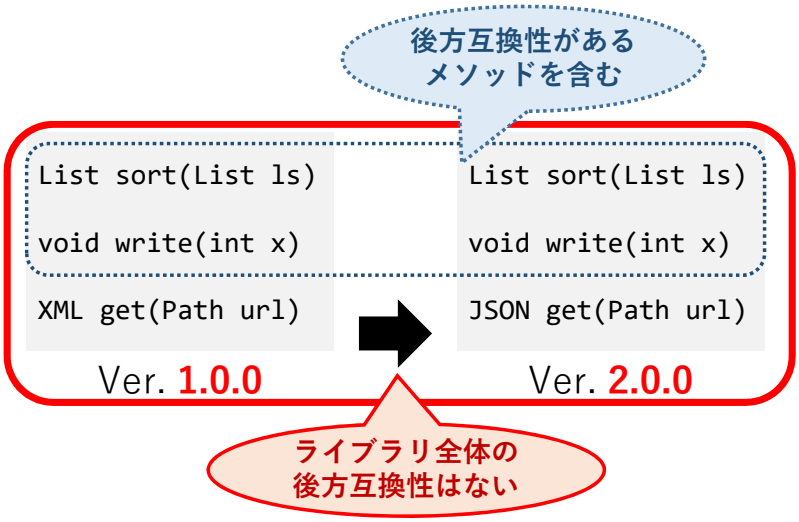

図 1　ライブラリ単位でのバージョニング

リリース日などのメタデータを付加するために，パッチバージョンまたはプレリリース版を表す文字列にプラス記号とドット区切りの識別子を続けることができる．識別子に利用できる文字はプレリリース版のそれと同じである．メタデータはバージョンの大小比較に用いない．

本研究では，プレリリース版およびメタデータを考慮しない．

2.2　既存のバージョニング手法における問題点

既存のバージョニング手法にはいくつかの問題点が考えられる．本節ではライブラリの開発者および利用者それぞれの立場において考えられる問題点について述べる．

2.2.1　ライブラリ開発者の立場における問題点

マイナーリリースやパッチリリースには本来後方互換性があるはずであるが，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ことが Raemaekers らの調査 [22] で明らかになっている．

従来の方法ではライブラリの開発者が手動でバージョンを決定している．そのため，ライブラリを複数人で開発している場合など，バージョンを決定する者が新しいリリースでの変更内容を完全には把握できておらず，このような誤りが発生すると考えられる．

2.2.2 ライブラリ利用者の立場における問題点

メジャーリリースには後方互換性のない変更だけでなく，後方互換性のある変更を含めることが認められている．そのため，図 1 のようにメジャーリリースにも後方互換性のあるメソッドを含んでいる可能性がある．ライブラリの利用者は，そのライブラリが持つすべての機能を利用するわけではなく，多くの場合はその一部を利用する．そのため，メジャーリリースを適用した際にソースコードを修正する必要があるとは限らない．

ライブラリに付けられたバージョンだけでは後方互換性のないメソッドを特定できないため，リリースノートなどの別の手段を用いる必要がある．リリースノートは必ずしも提供されているものではなく，また提供されていたとしてもその網羅性は保証されない．したがって，ライブラリのアップデートを適用した際の影響を調査するのに時間を要することが考えられる．実際，Raemaekers らが，ソフトウェアが依存しているライブラリは最新バージョンから数リリース前のものであるという調査結果を報告している [22].

2.3 リポジトリマイニング

ソフトウェア開発において，ソースコードの変更履歴を残すためにバージョン管理システムを用いてバージョン管理が行われることがある．バージョン管理システムが蓄積する変更履歴の情報をリポジトリという．これを解析して有用な情報を得ることをリポジトリマイニングという．近年ではオープンソースソフトウェア開発の広がりによって研究対象となるリポジトリの数が増加し，リポジトリマイニングに関する研究は盛んに行われている [15]．例えば，リポジトリメトリクスを用いて潜在的なバグを含むモジュールを予測する研究 [18] がある．また，ソースコードの変更履歴を利用して，ソースコードにどのような種類の変更がどのように行われているか，その変更の粒度はどの程度かなどを分析し，ソフトウェア開発プロセスの理解を試みる研究が行われている [21]．さらに，リポジトリマイニングに関する研究を支援するために，リポジトリホスティングサービスの 1 つである GitHub [5] に公開されているプロジェクトを収集し，リポジトリメトリクスやソースコードメトリクスなどを測定してデータセットを自動で構築するツールも開発されている [23].

本研究で提案する手法も，リポジトリのコミット履歴をもとにバージョニングを行うものであるからリポジトリマイニングの 1 種であるといえる．

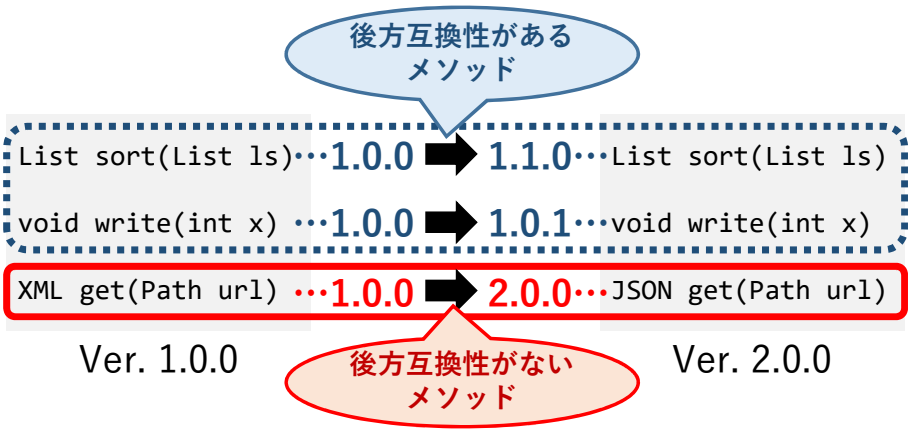

図 2　メソッド単位でのバージョニング

3　提案手法

本研究では，2.2 節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いメソッド（図の例では `get(Path)`）を具体的に提示できるようになる．

本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行う具体的な手順について述べる．

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に対して適用することができる．例えば Java 言語のメソッドや C++ 言語の関数などがこれにあたる．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．例えば Git [4] や Subversion (SVN) [2] などがこれにあたる．

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

以下，本節ではこれらの具体的な処理について説明する．

3.2.1 メソッドの追跡

1 つのソースコードファイルには一般に複数のメソッドの定義が含まれる．また，メソッドを一意に特定するために必要となる情報（シグネチャ）は変更されることがあり，さらにそのメソッドで行う一連の手続きが同時に変更されている可能性がある．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切り出し，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ことによりメソッドを「追跡」する必要がある．

提案手法では，前提としているバージョン管理システムのコミット履歴を利用してメソッドの追跡を行う．具体的には，隣接する 2 つのコミットの組 $(c_{\mathrm{old}}, c_{\mathrm{new}})$ について，新しい方のコミット c_{new} における各メソッド f_{new} と対応する古い方のコミット c_{old} におけるメソッド f_{old} を求める．対応するメソッドとは以下のいずれかを満たすメソッドである．

- シグネチャが同じメソッド
- シグネチャは異なるが，メソッドのボディが一定以上類似しているメソッド

なお，メソッド f_{new} に対して前者を満たすメソッドと後者を満たすメソッドが両方存在する場合は，シグネチャが同じ方を f_{old} とする．また，後者を満たすメソッドが複数存在する場合は最も類似しているメソッドを f_{old} とする．

後者の条件においてメソッドが「同じ」と判断する基準として用いる類似度としては，例えば以下のようなものが考えられる．

- 一致するバイト数
- 一致する行数
- 一致するトークン数
- 一致する手続きの数

ただし，以上の計算でメソッド f_{new} に対応するメソッド f_{old} が存在しない場合はコミット c_{new} において新たに追加されたメソッドとし，メソッド f_{old} に対応するメソッドが存在しない場合はコミット c_{new} において削除されたメソッドとする．

3.2.2 バージョンの決定

前項で求めたメソッドの対応関係から，コミット c_{old} から c_{new} にかけて変更されたメソッドを以下の 4 つに分類する．

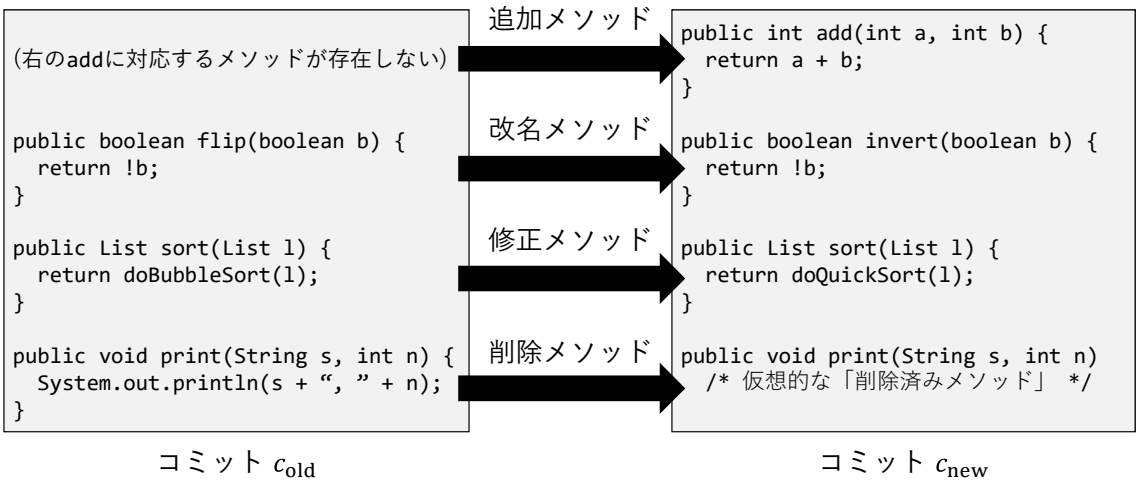

図 3　メソッドの対応関係の例

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，コミット c_{new} 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　メジャーバージョンを 1 増加させる．

修正メソッド　変更の内容によって以下のようにする．

- **後方互換性がない変更**　メジャーバージョンを 1 増加させる．
- **バグ修正ではない変更**　マイナーバージョンを 1 増加させる．
- **バグ修正である変更**　パッチバージョンを 1 増加させる．

削除メソッド　存在しないメソッドであるためバージョンは定められないが，削除前と同じシグネチャを持つ「削除済みメソッド」を仮想的に考え，メジャーバージョンを 1 増加させる．

上記では省略したが，セマンティックバージョニングの原則に従って，増加させる数字より右側にある数字はすべて 0 にする．

図 3 に以上のアルゴリズムで得られるメソッドの対応関係の例を示す．

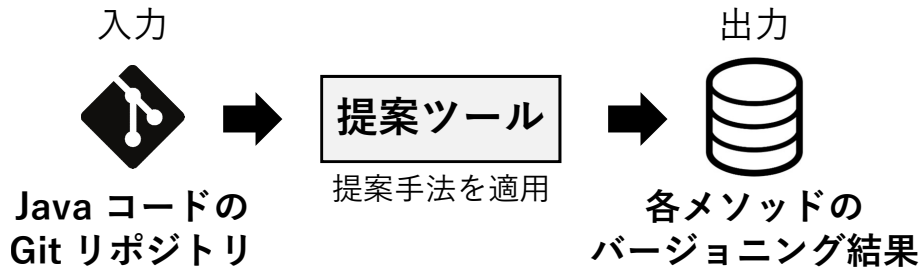

図 4　提案ツールのシステム概要

4　実装

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 [6] を採用した．

4.1　ツールの概要

提案ツールの概要を図 4 に示す．

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．メソッドレベルセマンティックバージョニングの結果はデータベースとして出力される．本ツールではデータベースとして SQLite 3 [14] を採用した．

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 コミット履歴の取得
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．

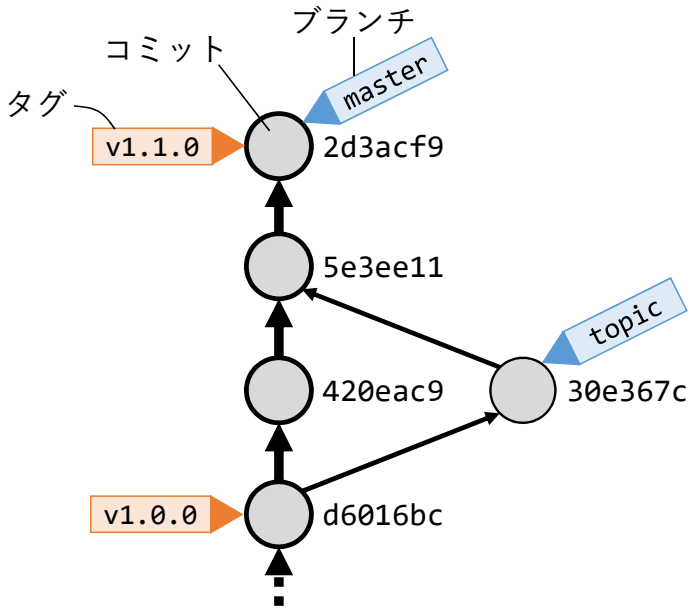

図 5　コミットグラフの例

Step 1: コミット履歴の取得

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）したりでき，一般にコミットの履歴は図 5 に示す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．

提案ツールでは，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 [16]．例えば，図 5 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 5 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．

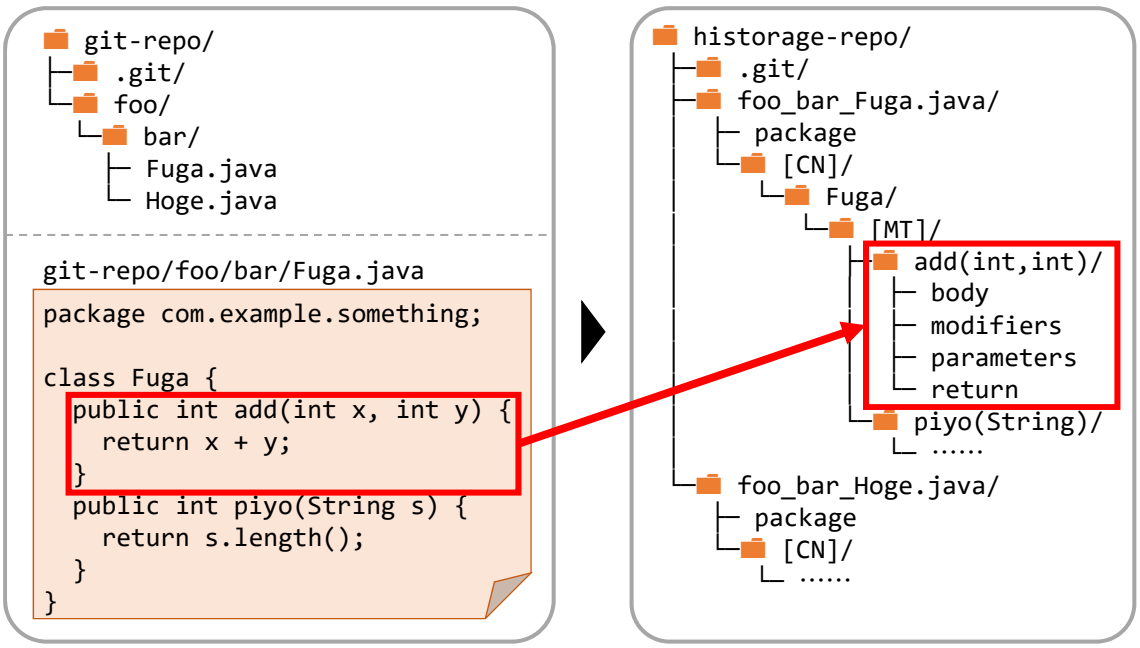

図 6　Git リポジトリとそれを変換した Historage リポジトリ

Step 2: メソッドの追跡

まず，入力された Git リポジトリを Historage リポジトリ [19] 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 とは，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 [19]．Git リポジトリを Historage リポジトリに変換するツールに Kenja [17, 8] がある．Kenja は，Git リポジトリを入力に取り，リポジトリ内のソースコードからメソッドの情報を抽出して図 6 のような構造のディレクトリを構築する．

GitHub [8] 上に公開されている Kenja が取得するメソッドの情報はメソッドの引数とボディ（図 6 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 [9] を返り値の型およびアクセス修飾子の情報（図 6 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．

次に，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，3.2.2 項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．アクセス修飾子が狭まるとは，そのメソッドを呼び出すことができる範囲が狭まることであり，より厳密に言えば以下の大小関係においてアクセス修飾子が小さくなるということである．

`public` > package-private > `protected` > `private`

ここで「package-private」とは，Java においてアクセス修飾子を指定しなかった場合にメソッドを呼び出すことができる範囲のことを指す．

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，Raemaekers らの調査 [22] で扱われている Maven Repository [10] 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が依存しているライブラリほど，セマンティックバージョニングに違反した際の影響が大きいと考えられるため，このような基準で対象ライブラリを選定した．

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 [12] 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 [1] 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の有無と比較するために，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 [7] 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.
```

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

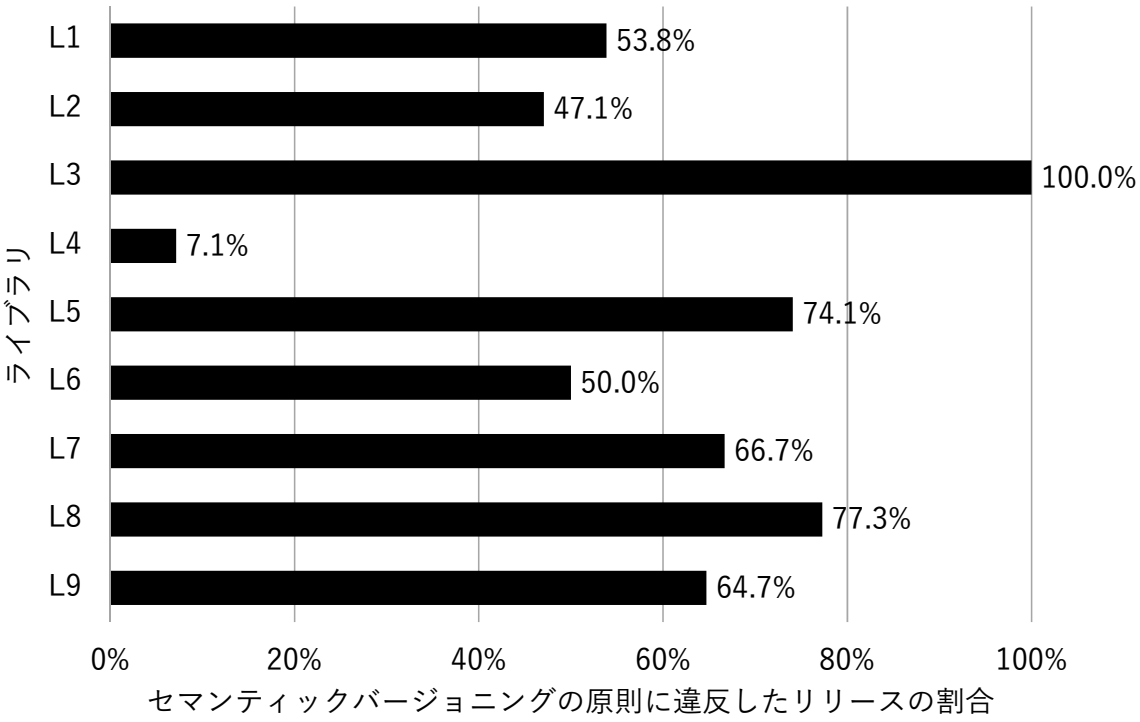

図 7　セマンティックバージョニングの原則に違反したリリースの割合

以下ではテストメソッド以外の public メソッドを単に public メソッドと呼ぶ.

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の原則に違反していると判断されたリリースの数とその総リリース数に対する割合を表 2 および図 7 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，ほとんどのライブラリで 4 割以上のリリースに誤ったバージョニングが行われていたことが分かった.

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．

この結果から，後方互換性のない変更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の割合は，Google Guava (L4) や Apache Commons Lang (L2) では少なく，Java Servlet API (L6) や Jackson Databind (L5) では多いことが分かる．

図 8 は，マイナーリリースおよびパッチリリースに含まれる後方互換性のないメソッドの数をリリースごとにプロットした箱ひげ図である．マイナーリリースやパッチリリースに 1 リリースあたり平均

表 2　セマンティックバージョニングの原則に違反したリリース

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%

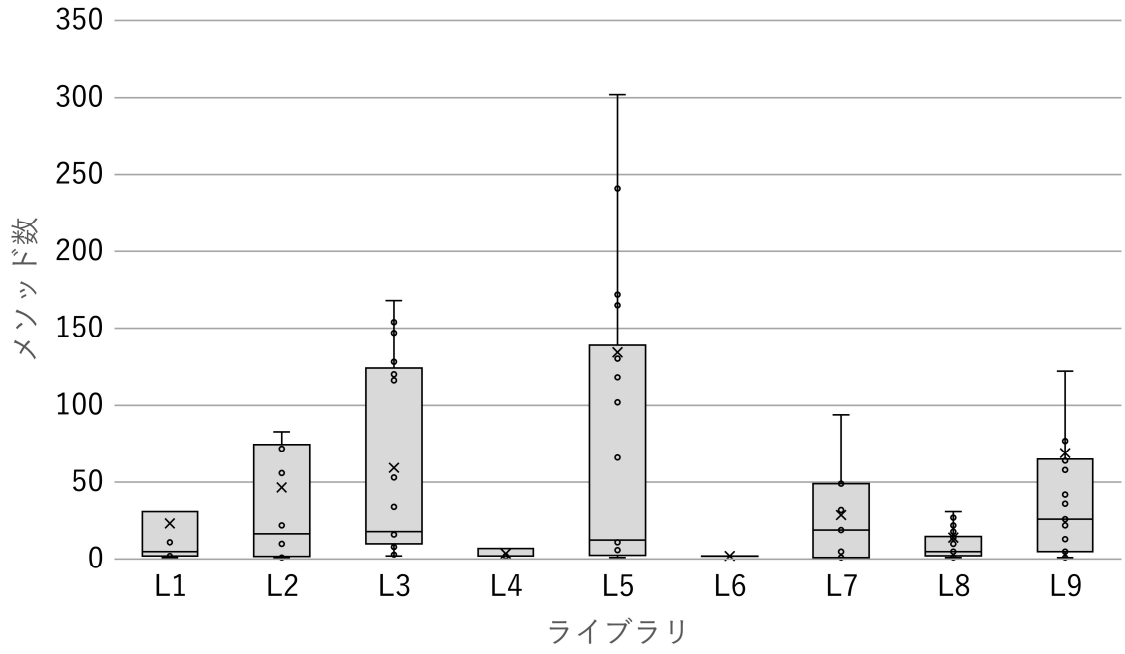

図 8　本来後方互換性があるはずのリリースに含まれている後方互換性のないメソッドの数

54.7 個のメソッドは後方互換性のないものであるという結果が得られた．

具体的に検出されたメソッドの例を示す．図 9 は本実験で検出された後方互換性のない変更の一例である．図 9 の変更は，Apache Commons Lang のバージョンが 2.1.0 から 2.2.0 へのマイナーリリースにおいて，org.apache.commons.lang.text パッケージの StrBuilder クラスのメソッド delete(char) が deleteAll(char) に改名されたというものである．

ライブラリバージョン 2.1.0

```
public StrBuilder delete(char ch) {
  for (int i = 0; i < size; i++) {
    if (buf[i] == ch) {
      int start = i;
      while (++i < size) {
        if (buf[i] != ch) {
          break;
        }
      }
      System.arraycopy(buf, i, buf, start, size - i);
      size -= (i - start);
    }
  }
  return this;
}
```

ライブラリバージョン 2.2.0

```
public StrBuilder deleteAll(char ch) {
  for (int i = 0; i < size; i++) {
    if (buffer[i] == ch) {
      int start = i;
      while (++i < size) {
        if (buffer[i] != ch) {
          break;
        }
      }
      int len = i - start;
      deleteImpl(start, i, len);
      i -= len;
    }
  }
  return this;
}
```

図 9　Apache Commons Lang において検出された後方互換性のない変更の例

6 考察

6.1 実験結果の考察

図 7 の結果から，利用数の多いほとんどのライブラリにおいて，4 割以上のリリースで誤ったバージョニングが行われていることが分かった．

プロジェクトの方針としてセマンティックバージョニングを行うことが示されている Apache Commons プロジェクトのライブラリ（Apache Commons IO および Apache Commons Lang）でも約半数のリリースで誤ったバージョニングが行われていた．これはバージョンを決定する際に，新たなリリースに後方互換性のない変更が含まれているかどうかを正しく確かめられていないことに起因していると考える．

6.2 先行研究との差異

Raemaekers らの調査 [22] では，後方互換性のない変更が行われた API を調べるために Clirr [3] というツールが用いられている．Clirr は 2 つの JAR (Java Archive) ファイルを入力すると，それらの間で後方互換性のないメソッドが検出される．JAR ファイルはリリースごとに生成，公開されるため，リリースごとにメソッドの対応付けが行われることになる．一方提案ツールはバージョン管理ツールを利用し，コミットごとにメソッドの対応付けを行っている．したがって，メソッドの追跡精度が Clirr より向上していることが期待される．しかし，本研究ではこれについて評価することができていないため，提案手法によるメソッドの追跡精度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．その結果，実験の対象とした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の原則に違反したリリースがあることがわかった．それにより，先行研究 [22] 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

本研究を行うにあたって理解あるご指導を賜り，暖かく励まして頂きました楠本真二教授に心より感謝を申し上げます．

本研究の全過程を通して熱心かつ丁寧なご指導を頂きました，肥後芳樹准教授に深く感謝を申し上げます．

本研究に関して有益かつ的確なご助言を頂きました，柗本真佑助教に深く感謝を申し上げます．

本研究を進めるにあたって様々な形で励ましやご協力を頂きました，楠本研究室の皆様に心より感謝を申し上げます．

最後に，本研究に至るまでの間，講義，演習，実験などで日々お世話になりました，大阪大学基礎工学部情報科学科の諸先生方にこの場を借りて心より御礼申し上げます．

参考文献

[1] Apache Commons – Versioning Guidelines. https://commons.apache.org/releases/versioning.html.

[2] Apache Subversion. https://subversion.apache.org/.

[3] Clirr. http://clirr.sourceforge.net/index.html.

[4] Git. https://git-scm.com/.

[5] Github. https://github.com/.

[6] JGit. http://www.eclipse.org/jgit/.

[7] JUnit. https://junit.org/.

[8] Kenja. https://github.com/niyaton/kenja.

[9] kenja-java-parser. https://github.com/niyaton/kenja-java-parser.

[10] Maven Repository. https://mvnrepository.com/.

[11] Osgi™ alliance – the dynamic module system for java. https://www.osgi.org/.

[12] ReleasePolicy · google/guava Wiki. https://github.com/google/guava/wiki/ReleasePolicy.

[13] Semantic Versioning 2.0.0. https://semver.org/spec/v2.0.0.html.

[14] SQLite Home Page. https://sqlite.org/index.html.

[15] Monden Akito, Ihara Akinori, and Matsumoto Kenichi. Mining software repositories. *Computer Software*, Vol. 30, No. 2, pp. 2_52–2_65, 2013.

[16] Scott Chacon and Ben Straub. *Pro git*. Apress, 2014.

[17] Kenji Fujiwara, Hideaki Hata, Erina Makihara, Yusuke Fujihara, Naoki Nakayama, Hajimu Iida, and Kenichi Matsumoto. Kataribe: A hosting service of historage repositories. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 380–383. ACM, 2014.

[18] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pp. 78–88, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Historage: fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pp. 96–100. ACM, 2011.

[20] Bauml Jaroslav and Brada Premek. Automated versioning in osgi: A mechanism for compo-

nent software consistency guarantee. In *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, pp. 428–435, Aug 2009.

[21] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, Vol. 31, No. 6, pp. 511–526, June 2005.

[22] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pp. 215–224. IEEE, 2014.

[23] 高澤亮平, 坂本一憲, 鷲崎弘宜, 深澤良彰. Repositoryprobe: リポジトリマイニングのためのデータセット作成支援ツール. コンピュータ ソフトウェア, Vol. 32, No. 4, pp. 4_103–4_114, 2015.