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

信学技報
IEICE Technical Report
SS2018-78 (2019-03)

ライブラリ進化への追随のためのソフトウェア修正の共有手法の提案

渥美　紀寿†　　桑原　寛明††

† 京都大学 学術情報メディアセンター
〒606-8501 京都府京都市左京区吉田本町
†† 南山大学 情報センター
〒466-8673 愛知県名古屋市昭和区山里町 18
E-mail: †atsumi.noritoshi.5u@kyoto-u.ac.jp, ††kuwabara@nanzan-u.ac.jp

あらまし　ソフトウェア開発においてライブラリのバージョン進化に伴なう更新作業は，開発中のソフトウェアが提供すべき機能の本質的な問題ではないため，疎かになりがちである．我々はこのライブラリのバージョン更新を支援するための手法を提案する．ライブラリのバージョン進化に従い，ライブラリが提供する API に互換性が保たれなくなることがある．我々はライブラリの各バージョンを解析することで，API の互換性が保たれているバージョンの範囲を提示可能にする．また，OSS における利用ライブラリのバージョン更新に関する更新履歴を解析することで，互換性のない API に関する修正方法を提示可能にする．

キーワード　ソフトウェア進化，ソフトウェア保守支援，プログラム解析

An Approach of Sharing Software Modification to Adopt Library Evolution

Noritoshi ATSUMI† and Hiroaki KUWABARA††

† Academic Center for Computing and Media Studies, Kyoto University
†† Center for Information and Communication Technology, Nanzan University
E-mail: †atsumi.noritoshi.5u@kyoto-u.ac.jp, ††kuwabara@nanzan-u.ac.jp

Abstract　In software development, the software maintenance task to update library version tends to be negligible because it is not related to the function provided by the software. We propose an approach to support adoptation of library update. Evolution of the library may lose compatibility of APIs provided by the library. By analyzing each version of the library, we make it possible to present a range of versions where API compatibility is maintained. Moreover, by analyzing the update history of the software that use the library, it is possible to present candidates of patch to migrate the incompatible API to the new API.

Key words　Software Evolution, Software Maintenance Support, Program Analysis,

1. はじめに

ソフトウェアの開発では，ライブラリやフレームワークなど様々な再利用資産 (以降ではこれらの再利用資産を単にライブラリと呼ぶ) が活用されている．ライブラリを利用してソフトウェアを構築することによって，すべてを自前で開発する必要がなく，効率良く開発することが可能となる．また，多くのソフトウェアで利用されているライブラリは十分なテストが行われていることから，それらを利用することで不具合を減らすことが可能となる [1].

ライブラリは開発対象のソフトウェアとは関係なく更新され，その更新には脆弱性修正やバグ修正などが含まれている．そのため，そのライブラリを利用しているソフトウェアそれぞれにおいて更新しなければ，ライブラリに存在した脆弱性やバグによる不具合が発生する可能性がある [2]．ライブラリの脆弱性やバグによる不具合を防ぐため，利用しているライブラリの更新に追随して，ビルドスクリプトやソースコードを修正する必要がある．しかし，ライブラリのバージョン移行は時間がかかる作業 [3] であり，放置されがちである．McDonnell らは Android を対象とした調査において，ライブラリの更新間隔が 3 ヶ月であるのに対し，それを利用したソフトウェアでのライブラリのバージョン移行間隔は 14 ヶ月であると報告している [4]．開発者は新しい機能を書くことや既知の問題を修正することを主な作業としていることが多く，その結果，潜在的な問題を解決するために重要ではあるが予防保守に当たるライブラリのバージョン移行作業は後回しにされがちである [5].

表 1　対象ライブラリ

groupId	artifactId	利用数
org.mockito	mockito-core	29
com.google.guava	guava	26
commons-codec	commons-codec	17
commons-io	commons-io	16
ch.qos.logback	logback-classic	16

一方，ライブラリの進化において，ライブラリが提供する API には後方互換性が維持されていないことがある．Mostafa らの研究では，多くのライブラリにおいて後方互換性が維持されていないこと，API ドキュメントやリリースノートには互換性に関する記述があまりされていないことを報告している [6]．Xavier らは API の変更の約 15 %が以前のバージョンとの互換性を維持していないことを報告している [7]．このように後方互換性が維持されないことがあるため，利用ライブラリのバージョン更新では，単純にバージョンを更新するだけでは不十分であり，ソースコードを修正する必要がある．

本稿ではこのような状況を改善するために，ライブラリのバージョン移行を支援する環境を構築し，ライブラリのバージョン移行作業を促進するための手法を提案する．

2. 関 連 研 究

本章ではライブラリの進化に伴なうソフトウェアの更新を支援に関する研究について紹介する．

Kalra らはライブラリバイナリの新旧両方のバージョンを実行し，そのトレースをすべて記録し，それらを比較して 2 つのライブラリバージョンに対する同じ API 呼び出しで意味的な類似性を判断する PULLUX を提案している [8]．この手法は動的情報を利用しているため，その動作を保証することが可能であるが，ライブラリを利用した機能に関するテストケースがない場合には正確な情報を得ることができない．また，新バージョンに対応していない場合，どのように修正すべきか，修正方法を提示することができない．

Mirhosseini らはプルリクエストとプロジェクトバッジを導入して開発者に古い依存関係を自動的に知らせることで新しいソフトウェアアップデートの可用性を常にチェックする手法を提案している [9]．7,470 の GitHub プロジェクトを対象とした調査では，プルリクエスト通知を使用しているプロジェクトは，ツールを使用していないプロジェクトと比べて，平均して 1.6 倍アップグレードされたことが報告されている．この手法は利用ライブラリに新バージョンが利用可能であることを通知するには有益であるが，それを利用するために修正が必要性や修正方法が提示されない．

本研究では，利用中のライブラリより新しいバージョンで互換性のあるより新しいバージョンの提示および修正方法が提示可能なより新しいバージョンとその修正方法の提示を可能にするための手法を提案する．

3. ライブラリの進化に伴なうソフトウェアの修正に関する調査

3.1 調 査 対 象

GitHub で公開されている Java プロジェクトで Maven による構成管理をしているプロジェクトのうち，2019 年 1 月 28 日の時点でスターの多い 100 のプロジェクトを対象とした．GitHub におけるスターは，プロジェクトの人気のための一般的で簡単にアクセスできる指標であるため [10]，我々はこの基準に基づき，対象を選択した．また，Java プロジェクトのうち，Maven による構成管理をしているプロジェクトを選択した理由はツールの都合である．

3.2 ライブラリの進化

多くの場合，バージョンは<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> のように付けられ，パッチバージョンの更新では機能の削除や追加等はほとんど行われず，バグ修正や脆弱性対応のための修正などで付けられる．マイナーバージョンの更新では機能強化が中心であり，API の互換性が維持されることが多く，メジャーバージョンの更新では大幅な変更があり，API の互換性が維持されないことが多い．特に API の互換性が維持されないような大きな変更は，新しい機能を実装する必要性，API をより単純化し，より少ない要素で，その保守性を改善したいという欲求から行われる [11]．

対象の 100 プロジェクトのうち 15 以上のソフトウェアで利用されているライブラリから無作為に選択した表 1 に示す 5 つのライブラリを対象とし，ライブラリが提供する API メソッドの数の変化を調査した．

各ライブラリのバージョンごとの API メソッドの数を表 2 に示す．'-' の行は前バージョンから削除された API メソッドの数を，'+' の行は前バージョンから追加された API メソッドの数を示している．本調査では，API メソッドのシグネチャだけでメソッドの同一性を判断しており，メソッドの実装は考慮していない．

`guava` の `23.1-jre` から `24.1.1-jre`, `logback-classic` から `0.8.1`, `0.9.30`, `0.9.30` から `1.0.13`, `1.1.11` から `1.1.13` の間は削除される API メソッド 数が数百と多いが，それ以外についてはあまり大きな差がない．一般的には，前述の通り，メジャーバージョンの更新時に機能の追加や削除が多く，マイナーバージョンの更新時には少ないと言われているが，実際にはメジャーバージョンの更新時でも変化が少ないこともあれば，マイナーバージョンの更新時でも変化が多いこともある．

3.3 ライブラリ利用ソフトウェアでの更新状況

3.2 節で対象とした 5 つのライブラリのうち，`commons-codec`, `commons-io`, `guava` を利用しているソフトウェアにおいて，どの程度ライブラリの進化に追随しているか調査を行った．

各ライブラリを利用しているソフトウェアに対し，pom.xml の変更履歴を追跡し，それぞれの変更履歴において指定しているライブラリのバージョンを記録することで，利用ライブラリのバージョンの変化を記録した．その結果を表 3 に示した．利用ライブラリのバージョン移行をまったくしていないソフト

表 2　対象ライブラリのバージョンごとの API メソッド数

mockito-core		guava		commons-codec		commons-io		logback-classic	
version	メソッド数	version	メドッド数	version	メソッド数	version	API 数	version	API 数
2.20.1	1,896	23.1-jre	9,365	1.7	435	2.2	964	0.8.1	417
−	3	−	240	−	0	−	0	−	162
+	16	+	345	+	5	+	27	+	761
2.21.0	1,909	24.1.1-jre	9,470	1.8	440	2.3	991	0.9.30	1,016
−	0	−	4	−	6	−	0	−	104
+	4	+	8	+	7	+	16	+	122
2.22.0	1,913	25.1-jre	9,474	1.9	441	2.4	1,007	1.0.13	1,034
−	0	−	11	−	0	−	0	−	14
+	1	+	32	+	74	+	81	+	46
2.23.4	1,914	26.0-jre	9,495	1.10	515	2.5	1,088	1.1.11	1,066
−	14	−	4	−	0	−	6	−	173
+	18	+	14	+	69	+	29	+	3
2.24.0	1,918	27.0.1-jre	9,505	1.11	584	2.6	1,111	1.2.3	896
共通	1,879	共通	9,106	共通	429	共通	964	共通	227

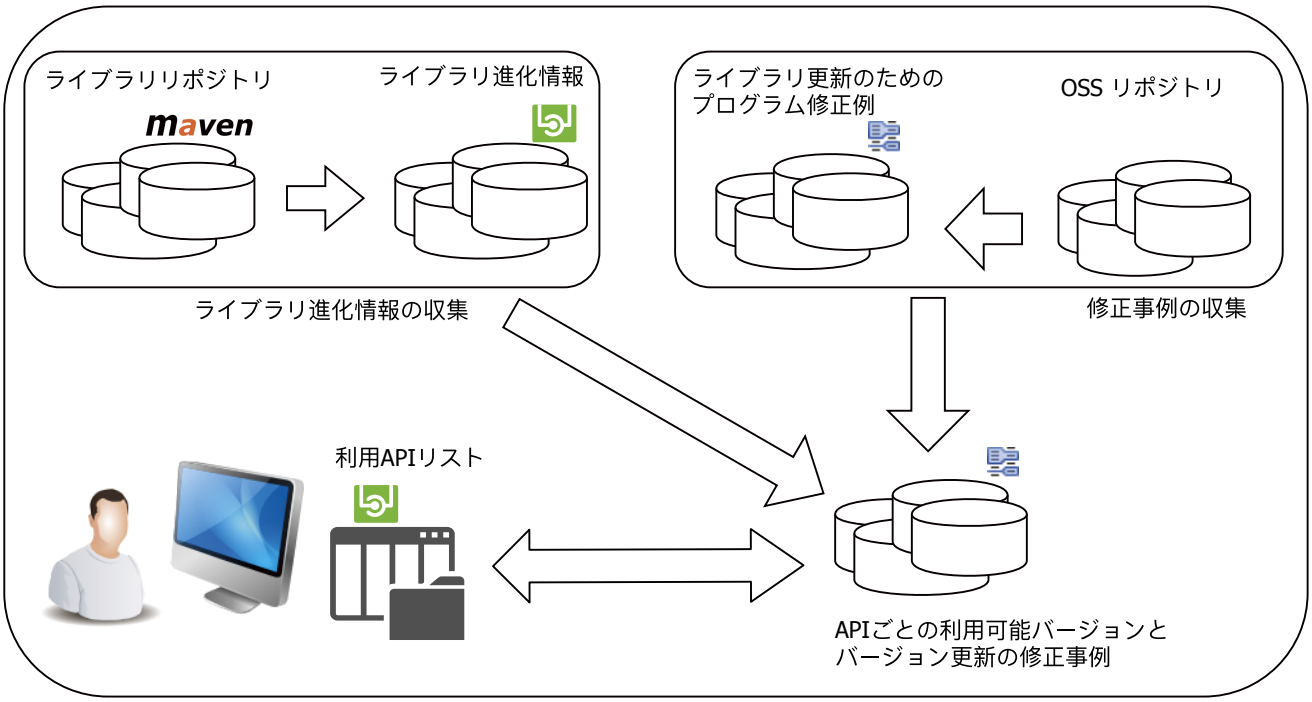

図 1　提案手法の概要

ウェアがある一方，適度なタイミングでバージョン移行しているソフトウェアが存在する．これらのソフトウェアからライブラリのバージョンを移行する際の修正方法を取得することによって，ライブラリのバージョン移行を支援することが可能である．

4. 提 案 手 法

3. 章の調査において，多くの既存のソフトウェアでは，利用しているライブラリのバージョンの移行はあまり行われておらず，移行していてもその頻度は低い．多様なライブラリを用いてソフトウェアが構築されており，それらのライブラリの更新状況を考慮しつつソフトウェアを維持していくことは非常に大変である．しかし，ライブラリの不具合や脆弱性対応等により，予期せぬ不具合を防ぐためにも利用ライブラリのバージョンを移行することは有用である．

我々はライブラリリポジトリから得られるライブラリの進化情報と OSS リポジトリから得られるライブラリの更新に伴なう修正事例に基き，開発中のソフトウェアで利用しているライブラリの更新に伴ない必要となる修正方法を，ライブラリのバージョンごとに提示する手法を提案する．提案手法の概要を図 1 に示す．

ライブラリは Maven Central Repository (Java)，Gem Repository (Ruby)，PyPI (Python) 等で公開されており，これらのリポジトリ (本稿では，このリポジトリをライブラリリポジトリと呼ぶ) に登録されているライブラリをバージョンごとに解析し，公開 API を各ライブラリのバージョンごとに蓄

表 3　ライブラリ利用ソフトウェアにおけるライブラリバージョン変化

commons-codec (最新 ver.1.11)

ソフトウェア名	ライブラリバージョン
jsonschema2pojo	1.9
disconf	1.9
motan	1.4
titan	1.4, 1.7
neo4j	1.7, 1.9, 1.11
flink	1.3, 1.10
rocketmq	1.9
canal	1.9
pinpoint	1.10

commons-io (最新 ver.2.6)

ソフトウェア名	ライブラリバージョン
checkstyle	2.6
jsonschema2pojo	2.4
spring-mvc-showcase	2.0.1, 2.5
disconf	2.4
titan	2.0.1, 2.1, 2.3
neo4j	1.4, 2.4, 2.6
flink	2.4
canal	2.4
incubator-druid	2.0.1, 2.4, 2.5
pinpoint	1.4, 2.5

guava (最新 ver.27.0.1)

ソフトウェア名	ライブラリバージョン
pinpoint	17.0, 19.0, 20.0
miaosha	18.0
disconf	19.0-rc2
motan	18.0
Mybatis-PageHelper	17.0, 19.0
rocketmq	19.0
canal	r09, 18.0
auto	14.0.1, 15.0-rc1, 15.0, 16.0-rc1, 16.0, 16.0.1, 23.5-jre
webmagic	13.0.1, 15.0
guice	16.0.1, 18.0, 19.0, 23.6-android, 25.1-android

積する．OSS は GitHub 等で公開されており，これらのリポジトリに登録されているソフトウェアを対象にビルドファイルの改訂履歴を基に，利用しているライブラリの更新とそれに対応する修正事例をライブラリごとに蓄積する．ライブラリから得られる進化情報とライブラリを利用しているソフトウェアから得られた修正情報から，各ライブラリのそれぞれのバージョンごとの修正方法としてまとめる．これを開発者に提示することによって，ライブラリの更新に伴なう修正を支援する．

4.1　ライブラリ進化情報の収集

ライブラリは新機能の提供，セキュリティ対策，不具合修正などの目的でバージョンが更新され，これらの過程でリファクタリングなどが行われ，不要な API の削除や既存 API に変化が生じることがある．本章ではその変化を収集する方法について述べる．

ライブラリリポジトリでは，ライブラリを一意に特定するために識別子とそのバージョンが用いられている．具体的には Java ライブラリの公開に広く利用されている Maven 形式のリポジトリでは， groupId と artifactId によってライブラリが一意に識別され，Ruby や Python では名前によって識別される．ライブラリ利用者はビルドファイル (Java では pom.xml や build.sbt, build.gradle など，Ruby では Gemfiles，Python では Pifiles) でライブラリとバージョンを指定することによって，特定のバージョンのライブラリを用いてビルドすることが可能である．

提案手法では，このライブラリリポジトリから，個々のライブラリについて提供されている全バージョンを取得し，個々のバージョンで公開されている API として提供されるメソッドのシグネチャを解析し，シグネチャごとに提供しているバージョンを記録する．

表 4 に Java のライブラリ `commons-io` の API リストの例を示す．

4.2　ライブラリ更新に伴なう修正事例の収集

多くの OSS ソフトウェアが GitHub などで公開されており，更新履歴が蓄積されている．これらのソフトウェアでは，多くのライブラリが利用されており，その更新履歴にはライブラリのバージョン移行に関する情報が含まれていると考えられる．利用しているライブラリはビルドファイルに記述されることが多く，それらの更新履歴からライブラリのバージョン移行が行われた履歴を得ることが可能である．また，API メソッドに変更がある場合には，ビルドファイルの更新とともにソースコードが変更される．このソースコードの差分から，修正が必要な API メソッドについて修正前の API メソッドと修正後の API メソッドを取得し，修正事例として記録する．これにより API メソッドごとにライブラリの更新に必要な情報を提示することが可能となる．

4.3　API ごとの修正方法

4.1 節で述べたライブラリ進化情報により，各 API メソッドが利用可能なバージョンの範囲がわかる．4.2 節で述べた修正事例では，ソースコードの修正が必要な API メソッドについて，利用ライブラリの特定のバージョン α から特定のバージョン β への移行に必要となる修正方法が得られる．API メソッドの変化は一般的にはシグネチャに変更がない限り，その動作は同じである．そのため，事例から得られた修正方法が適用可能なバージョンは修正前の API が利用可能なバージョン範囲から修正後の API が利用可能なバージョン範囲となる．これらの情報を API ごとの修正方法として蓄積しておき，開発中のソフトウェアで利用しているライブラリのバージョン移行で必要となる情報を提示する．

ただし，メソッドのシグネチャが同じでもその API が提供する機能の動作に互換性がない場合がある [6]．我々の提案手法ではこの情報を得ることが困難である．しかし，ライブラリの更新に関する情報が多数蓄積されることによって，そのような場合の対応方法が蓄積され，それらを分析することでより詳細な情報を提供することが可能となる．

表 4　commons-io の API リスト

API	バージョン					
	...	2.2	2.3	2.4	2.5	2.6
org.apache.commons.io.ThreadMonitor.run()V	...	✓	✓	✓	✓	✓
org.apache.commons.io.Charsets.<init>()V	...		✓	✓	✓	✓
org.apache.commons.io.IOUtils.close(Ljava/net/URLConnection;)V	...			✓	✓	✓
org.apache.commons.io.IOUtils.writeChunked([CLjava/io/Writer;)V	...				✓	✓
org.apache.commons.io.input.ObservableInputStream.read()I	...					✓

表 5　対象ソフトウェア

ソフトウェア	利用ライブラリ
motan	commons-codec 1.4
rocketmq	commons-codec 1.9
jsonschema2pojo	commons-codec 1.9, commons-io 2.4
canal	commons-codec 1.9, commons-io 2.4
spring-mvc-showcase	commons-io 2.5
solo	commons-codec 1.10
symphony	commons-codec 1.10
incubator-shardingsphere	commons-codec 1.10
Mybatis-PageHelper	guava 19.0
retrofit	guava 19.0
java-design-patterns	guava 19.0
JCSprout	guava 22.0

本研究では，ビルド可能なできるだけ新しいライブラリのバージョンを提示すること，およびより新しいバージョンを適用する際に修正が必要な API メソッドを提示することによって，利用ライブラリのバージョン更新に必要なコストを見積ることを可能にする．これにより，利用ライブラリのバージョン移行の促進を図る．

5. 実　　験

提案手法によるライブラリ進化に伴なうソフトウェア修正方法の提示がどの程度可能か, OSS を対象に実験を行った．実験対象は 3. 章で調査したライブラリ のうち guava, commons-codec, commons-io を利用し，ビルドできた 12 のソフトウェアである (表 5)．本実験では，ライブラリが提供する API メソッドおよび，ライブラリを利用しているソフトウェアにおける参照 API メソッドの抽出には，JDK で提供される tools.jar を利用して開発したツールを用いた．本ツールはクラスファイルからメソッド定義および参照しているメソッドを抽出するツールであるため，実験環境において，ビルドできなかったソフトウェアは対象から除外している．

5.1 実 験 内 容

実験では以下の項目について確認した．

実験 1　新しいバージョンに移行するに当たり，ビルドファイル (pom.xml) で指定しているバージョンを上げるだけでビルドおよびテストが成功するソフトウェアがどの程度あるか．

実験 2　新しいバージョンに移行するに当たり，ソースコードの修正が必要な場合，既存のソフトウェアから対応する修正方法を提示可能か

5.2 実 験 結 果

実験 1 の結果

motan，rocketmq，jsonschema2pojo，canal，solo，symphony，incubator-shardingsphere，Mybatis-PageHelper，retrofit の 9 つについては，pom.xml のバージョン指定を最新のライブラリのバージョン (commons-codec 1.11，commons-io 2.6，guava 27.0.1-jre) に変更するだけで，ビルドが成功し，テストも成功した．

実験 2 の結果

java-design-patterns, JCSprout, spring-mvc-showcase の 3 つについては，利用している API メソッドの中に新しいバージョンのライブラリには存在しない API メソッドがあり，ソースコードの修正が必要であった．

spring-mvc-showcase では参照している API メソッド 409 種類中 2 種類のメソッドが, java-design-patterns では 3,862 種類中 532 種類のメソッドが，JCSprout では 18 種類中 2 種類のメソッドが新しいバージョンのライブラリに存在していなかった．実験対象のソフトウェアにおいてこれらの API メソッドに対する修正は存在しておらず，修正に必要な情報を提示することができなかった．

5.3 考　　察

実験 1 では，ビルドおよびテストが成功したが，プロジェクトで用意されているテストケースが利用しているライブラリの機能をどの程度カバーしているか不明であるため，不具合を含んでいないことを確証することはできない．しかし，ライブラリのバージョン移行において，テストが成功することは移行する動機付けになると考えられ，ライブラリのバージョン移行を促進するという目的において，この情報を提示できることは有用である．また，今回の実験対象からは得られないが，特定のライブラリのバージョンまでの移行実績を他のソフトウェアから得ることができれば，その情報を提示することによってライブラリのバージョン移行における不具合への不安を除くことが可能となる．

実験 2 では，ライブラリのバージョン移行に必要なソースコードの修正事例を提示することができなかったが，対象プロジェクトが少なかったことが原因の 1 つである．また，ライブラリのバージョン間において，提供している API メソッドのシグネチャの変化が大きい場合，対応する修正事例が少なく，

かつ修正事例におけるソースコードの差分も大きくなるため，ライブラリのバージョン移行のコストを下げることは難しい．

現状では利用ライブラリのバージョン移行はあまり行われておらず，ライブラリ更新に伴なうソースコードの修正情報を十分に取得することはできない．提案手法により，ライブラリの更新が促進されるに従ってこれらの情報を得ることが可能となる．新バージョンのライブラリに移行した場合に，発生した不具合等は現状ではあまり蓄積されていないが，ライブラリ更新作業が頻繁に行われるようにすることで，これらの情報が蓄積される．Suwa らの調査ではライブラリのバージョン移行後に前のバージョンにロールバックすることがあることが報告されている [12]．ロールバックした事例はバージョン移行に注意が必要であることを示すことができる．これらの情報はそのライブラリを利用している他の開発者にも有益な情報である．また，これらの情報をライブラリ開発者に提供することで，ライブラリのリリースノートや API ドキュメントへの反映を促すことが可能となる．

6. まとめと今後の課題

本研究では，ライブラリの進化に伴なうクライアントソフトウェアの修正を促進するために，ライブラリが提供する API の変化および既存のソフトウェアにおける修正方法を共有する手法を提案した．ライブラリの進化において，API メソッドのシグネチャの変化や削除が行われるが，その数はマイナーバージョンの更新では少ないこと，メジャーバージョンではマイナーバージョンの更新と比較すると多いが，クライアントソフトウェアで利用している API に変化がある数はそれほど多くないことがわかった．既存のソフトウェアでは，利用しているライブラリのバージョンの更新は一定数行われていることがわかった．

本研究の調査および実験では，GitHub に登録されているスターの多い 100 のプロジェクトを対象にしたが，対象プロジェクトの数が少なく，ライブラリ更新に必要となる情報を十分に得ることができなかった．そのため，より多くのプロジェクトを対象とした実験が必要である．対象プロジェクトを広げることで，ライブラリを利用したソフトウェアが多くなり，新しいバージョンのライブラリに更新する際の修正方法をより多く得られる可能性がある．提案手法を基にライブラリの収集および既存のソフトウェアでの利用ライブラリの更新状況の調査を行ったが，これらを自動化することで日々更新されるライブラリおよびそれらを利用したソフトウェアの情報を収集する必要がある．

提案手法では，API メソッドのシグネチャが同じ場合，動作も同じであることを仮定しているが，実際には動作が変更されることがある．そのような場合に他のソフトウェアでの移行実績を示すことで，移行に伴なう不具合への不安を減らすことが可能である．また，API メソッドの中には将来廃止予定で `@Deprecated` 注釈が付けられていることがある．互換性があるとして提供する情報の精度を高くするためには，これらの情報を解析する必要がある．また，`@Deprecated` 注釈が付けられたソフトウェアでは，そのメソッド内部で新しい API を用いた処理が記述されることが多く，それらを開発者に提示することは開発者にとって有益である．

謝辞 本研究の一部は科研費 (#15K15973, #15H02683, #18K11241) の助成を受けた．

文　献

[1] J. Visser, A. vanDeursen, and S. Raemaekers, “Measuring software library stability through historical version analysis,” Proceedings of the 2012 IEEE International Conference on Software Maintenance, pp.378–387, 2012.

[2] M. Cadariu, E. Bouwers, J. Visser, and A. vanDeursen, “Tracking known security vulnerabilities in proprietary software systems,” Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, pp.516–519, 2015.

[3] S. McIntosh, B. Adams, T.H.D. Nguyen, Y. Kamei, and A.E. Hassan, “An empirical study of build maintenance effort,” Proceedings of the 2011 33rd International Conference on Software Engineering, pp.141–150, 2011.

[4] T. McDonnell, B. Ray, and M. Kim, “An empirical study of api stability and adoption in the android ecosystem,” Proceedings of the 2013 IEEE International Conference on Software Maintenance, pp.70–79, 2013.

[5] 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, 2018.

[6] S. Mostafa, R. Rodriguez, and X. Wang, “Experience paper: A study on behavioral backward incompatibilities of java software libraries,” Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp.215–225, 2017.

[7] L. Xavier, A. Brito, A. Hora, and M.T. Valente, “Historical and impact analysis of api breaking changes: A large-scale study,” Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, pp.138–147, 2017.

[8] S. Kalra, A. Goel, D. Khanna, M. Dhawan, S. Sharma, and R. Purandare, “Pollux: Safely upgrading dependent application libraries,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.290–300, 2016.

[9] S. Mirhosseini and C. Parnin, “Can auto mated pull requests encourage software developers to upgrade out-of-date dependencies?,” Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp.84–94, 2017.

[10] H. Borges, A. Hora, and M.T. Valente, “Understanding the factors that impact the popularity of github repositories,” Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution, pp.334–344, 2016.

[11] A. Brito, L. Xavier, A. Hora, and M.T. Valente, “Why and how java developers break apis,” Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, pp.255–265, 2018.

[12] H. Suwa, A. Ihara, R.G. Kula, D. Fujibayashi, and K. Matsumoto, “An analysis of library rollbacks: A case study of java libraries,” Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference Workshops, pp.63–70, 2017.