Vol.2025-SE-219 No.25
2025/3/4

ライブラリ進化に追随するためのソースコード修正の網羅的な収集に向けて

桑原 寛明[1]　渥美 紀寿[2]

概要：本稿では，ライブラリのバージョン更新に対応するためのソースコード修正を収集する手法を提案する．ライブラリの進化に伴い提供される API が変更された場合，API を利用するソースコードを変更する必要がある．変更方法が必ずしも文書化されているとは限らず，ライブラリ自体やライブラリを利用する OSS の調査が必要となる場合もある．提案手法では，OSS を対象とし，利用ライブラリのバージョンが更新されたコミットにおいて，更新されたライブラリの API に関わるソースコード修正を収集する．これにより，互換性が失われた API を利用するソースコードの修正方法を開発者に提示可能にする．

Towards Exhaustive Collection of Source Code Modifications for Adopting Library Evolution

1. はじめに

ソフトウェア開発では様々な再利用資産を活用することで，すべてを一から自前で開発する必要はなくなり，効率的に開発を進めることができる．特に，プログラミングにおいてはライブラリやフレームワーク（以下，まとめてライブラリと呼ぶ）が利用される．多くのソフトウェアで利用されるライブラリは十分にテストされていることが期待できるため，そのようなライブラリを積極的に利用することで不具合の発生を減らすことができる [1].

一般に，ライブラリの更新はそのライブラリを利用するソフトウェアの開発とは独立して行われる．更新にはバグの修正や脆弱性の修正が含まれるため，古いライブラリの利用を継続すると修正されたバグや脆弱性が残ったままとなる [2]．これらのバグや脆弱性による不具合を防ぐには，ビルドスクリプトやソースコードを修正してライブラリの更新に追随する必要がある．しかし，利用ライブラリを新しいバージョンのものに入れ替えるバージョン移行はコストがかかる作業であり [3]，放置されたり後回しにされやすい．McDonnell らによる Android を対象とした調査では，Android API の更新期間が 3 ヶ月である一方で，Android アプリケーションが利用する Android API のバージョン移行間隔は 14 ヶ月であることが報告されている [4]．開発者は新しい機能の実装や既知の問題の修正に集中していることが多く，その結果として，潜在的な問題の解決には重要であるが予防保守にあたる利用ライブラリのバージョン移行は後回しにされやすい [5].

ライブラリの進化において，ライブラリが提供する API の後方互換性が維持されなくなることがある．Mostafa らは，多くのライブラリにおいて API の後方互換性が維持されていないこと，API のドキュメントやリリースノートに互換性に関する情報が記載されることは少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]．このようにライブラリの進化に伴って API の互換性が維持されないことがあり，そのような API を利用している場合，利用ライブラリのバージョン移行時にソースコードの変更が必要となる．互換性について文書化されていなければ，当該ライブラリのソースコードや当該のバージョン移行を行ったソフトウェアのソースコードを調査する必要が生ずる．

本稿では，API の後方互換性が維持されておらず，その非互換性が文書化されていない場合であっても，利用ライブラリのバージョン移行を支援するための手法を提案する．当該のバージョン移行を行ったソフトウェアにおいて

1 南山大学 理工学部
Faculty of Science and Technology, Nanzan University
2 京都大学 学術情報メディアセンター
Academic Center for Computing and Media Studies, Kyoto University

© 2025 Information Processing Society of Japan

は，その移行時に API の非互換性に対応するための修正を行なった可能性が高い．そこで，OSS プロジェクトを対象として，後方互換性が維持されないライブラリ進化に追随するためのソースコード修正，特に非互換 API 呼び出しの修正を網羅的に収集する手法を提案し，予備実験の結果を報告する．

なお，我々は本研究に先行して実施した予備調査について過去に報告している [8]．この予備調査では GitHub で公開されている OSS を対象として，

- 5 種類のライブラリについて更新時に削除および追加されるメソッドの数
- 3 種類のライブラリについてそれを利用するいくつかのソフトウェアにおけるバージョン移行の実施状況

を調査した．さらに，12 個のソフトウェアにおいて 3 種類のライブラリのバージョン移行を実施し，

- 9 個のソフトウェアではバージョン移行後のビルドとテストに成功
- 3 個のソフトウェアでは利用ライブラリの更新で削除されたメソッドを呼び出していたためビルドに失敗
- 削除されたメソッドに対応する修正例が調査対象の中には不存在

という結果が得られた．予備調査の対象としたライブラリやソフトウェアはごく少数にとどまっている．本研究では，非互換 API 呼び出しに対する修正の収集をより大規模に実施することを目指す．

2. 関連研究

ライブラリの更新に関する調査研究として，ライブラリ更新の目的や更新タイミングに関して分析した研究がこれまでに行われている．Fujibayashi ら [9] は，ライブラリのリリースサイクルと利用ライブラリのバージョン移行の関係を調査した．23 のライブラリのリリースサイクルと 415 の Apache Software Foundation のクライアントプロジェクトを対象とした調査の結果，リリースサイクルが短いライブラリは長いライブラリと比較して早いタイミングでバージョン移行が行われることがわかった．Kula ら [5] は，利用ライブラリの更新について実証的に分析し，開発者が利用しているライブラリを新しいバージョンにほとんど入れ替えていないことを明らかにした．加えて，ライブラリの新バージョンのリリースとセキュリティ勧告に対する開発者の反応を理解するために 8 つのケーススタディを実施し，開発者がセキュリティ勧告に反応しない主な理由は，脆弱なライブラリを認識していないためであることを明らかにした．Wang ら [10] は，ライブラリの利用，更新，リスクについて実証的な調査研究を行っている．806 の OSS プロジェクトと 13,565 のライブラリについて，ライブラリの利用状況とライブラリの更新状況を分析した．さらに，806 の OSS プロジェクトと 544 のセキュリティバグについてライブラリのリスク分析を行った．調査の結果，古いバージョンのライブラリが多くのプロジェクトで採用されており，最新バージョンとの隔たりも大きいことがわかった．その他，セキュリティ上の不具合があるバージョンのライブラリを利用していることが判明したプロジェクトに対して，そのライブラリのバージョンとセキュリティバグ，安全なバージョンについて報告した結果についても述べている．これらの研究は，ライブラリのバージョン更新に対するクライアントソフトウェアの状況について調査しているが，我々はクライアントソフトウェアにおける修正方法を収集することを目的としている．

異なるライブラリ間での移行に関する研究がこれまでに行われている．Alrubave ら [11] は，入力された GitHub プロジェクトのリストに基づき，各リポジトリをクローンし，異なるライブラリへの移行に関する履歴をマイニングすることによって，ライブラリ間の移行関係を検出するツールを提案した．Teyton ら [12] らは，OSS のプロジェクトの開発履歴において，あるライブラリを別のライブラリに移行した変更履歴からそれぞれのライブラリが提供するメソッド間の対応関係を自動抽出する手法を提案した．Apache Commons と Google Guava の間で移行を行った OSS プロジェクトからメソッド間の対応関係を抽出し，高い精度で検出できたことを示した．He ら [13] も同様に既存の開発履歴から異なるライブラリ間の移行情報をマイニングし，精度を向上させるために 4 つのメトリクスを提案している．これらの研究は異なるライブラリ間での移行の支援に有用であるが，本研究では，同一ライブラリの新しいバージョンに対応するための修正方法を収集することを目的としている点で異なる．

Huang ら [14] は，ライブラリの更新において削除された API に対する代替 API を自動検出するための手法を提案し，RepFinder を実装した．RepFinder は，最初に非推奨 API であることを表すメッセージを新しいライブラリのソースコードから探し，そこで実装されたコードを候補として抽出する．次に，API が他のクラスに移動される可能性があるため，類似クラスで同様の名前の API を検出する．最後に，Wang[10] らの手法を利用し，他のライブラリから類似 API を検索することによって代替 API を検出する．我々は，ライブラリを利用しているクライアントソフトウェアの開発履歴から，ライブラリの更新に対応するための修正方法を収集する手法を提案する．

3. 提案手法

3.1 概要

利用ライブラリのバージョン移行時に，新旧のバージョン間で後方互換性が維持されない API をソースコード中で利用していた場合，同等の機能を他の API を利用して実現するためのソースコード修正が同時に行われる可能性が

© 2025 Information Processing Society of Japan

高い．そこで，利用ライブラリのバージョン移行と同時に行われたソースコード修正の中から，ライブラリの更新に伴って削除された API を呼び出すコード周辺の修正を抽出する．多数のプロジェクトから抽出し，ライブラリの名前，新旧のバージョン，非互換 API，ソースコード修正をデータベース化することで，非互換 API を利用するソースコードの修正方法を得ることが容易になる．

本稿では，1 つのプロジェクトから目的とするソースコード修正を抽出する手法を構築する．適用対象のプロジェクトは，以下の条件を満たすとする．

- ソースコードなどのファイルの変更を追跡できる
- 利用ライブラリとそのバージョンが管理されており，かつ変更を追跡できる

1 つ目の条件は，利用ライブラリのバージョン移行に伴うソースコードなどのファイルの変更を取り出すため，2 つ目の条件は，利用ライブラリのバージョン移行時の新旧のバージョンを特定するために必要である．

3.2 手順

以下の手順で利用ライブラリのバージョン移行に伴う非互換 API 利用コードの修正をプロジェクトから抽出する．全体像を図 1 に示す．

(1) 利用ライブラリのバージョン移行が行われたコミット（以下，バージョン移行コミットと呼ぶ）と，それぞれコミットにおけるバージョン移行の対象ライブラリとその新旧のバージョンを特定する
(2) (1) で特定されたライブラリの新旧バージョン間で削除されたメソッドをすべて列挙する
(3) バージョン移行コミットとその親コミットにおいて，それぞれソースコード中の各メソッド定義毎に，メソッド本体内で呼び出されているメソッドのリストを抽出する
(4) バージョン移行コミットにおいて，各メソッド定義毎に，親コミットにおける同一のメソッド定義と (3) で抽出されたリストを比較し，削除された被呼び出しメソッドを特定する
(5) (4) で特定されたメソッドが (2) のメソッド群に含まれていれば，利用ライブラリのバージョン移行に伴って非互換 API の呼び出しが修正されていると判断する
(6) (5) の該当箇所のソースコード差分を生成する

以下では，ソースコードなどファイルのバージョン管理に git を，利用ライブラリなどの構成管理に Maven を利用している Java プロジェクトを前提として，各手順の詳細を述べる．ただし，ツールやプログラミング言語によらず原則は同じである．

3.3 バージョン移行の特定

手順 (1) で，利用ライブラリのバージョン移行が行われたコミットを特定する．Maven の場合，図 2 のように pom.xml の `dependency` 要素に利用するライブラリを指定する．`groupId` と `artifactId` の組でライブラリが一意に特定され，`version` でライブラリのバージョンが指定される．そこで，pom.xml が変更されたコミット（以下，POM 変更コミットと呼ぶ）を最初に列挙し，列挙された各コミットについて，時系列順に直前のコミットにおける `groupId` と `artifactId` が同じ `dependency` 要素と `version` が異なるコミットをバージョン移行コミットとする．

なお，POM 変更コミットは主ブランチ（一般に main あるいは master が多い）のみから列挙する．異なるブランチのコミットをまとめて時系列順に並べることには意味がなく，多くのブランチはいずれ主ブランチにマージされるためである．

3.4 非互換 API の抽出

バージョン移行の対象ライブラリと新旧バージョンが特定できたので，手順 (2) で，新旧のバージョン間で削除されたメソッドを非互換 API として抽出する．Maven を利用している Java プロジェクトの場合，対象ライブラリの新バージョンと旧バージョン両方の JAR ファイルをダウンロードし，それぞれに対して，ライブラリ内で定義されるすべてのメソッドのシグネチャを抽出する．旧バージョンに存在し新バージョンに存在しないメソッドを削除されたメソッドとする．

API が対象であるため，抽出対象はパッケージ外に公開される `public` および `protected` なメソッドに限る．非推奨を意味する`@Deprecated` アノテーションが付けられたメソッドも削除されたメソッドとして扱う．オーバーロードされたメソッドを区別するため，シグネチャは，メソッドの完全修飾名，引数の型，返り値の型から構成される．

3.5 非互換 API 呼び出しを削除する修正の特定

バージョン移行コミットにおける非互換 API が特定できたので，この非互換 API の呼び出しを削除する修正を検索する．利用ライブラリのバージョン移行後のソースコードに非互換 API の呼び出しが残っているとコンパイルできないため，非互換 API の呼び出しの削除はバージョン移行コミットで行われる可能性が高い．そこで，バージョン移行コミットのソースコードを検索対象とする．

手順 (3)(4)(5) で，プロジェクト内で定義されている各メソッドについて，その中で呼び出されているメソッド群をバージョン移行コミットとその親コミットの間で比較する．親コミットに存在しバージョン移行コミットに存在しないメソッド呼び出しの中に非互換 API の呼び出しが含まれていれば，そのメソッド定義において非互換 API の呼び出しが修正されたとみなし，手順 (6) で該当箇所のソースコードの差分を生成する．

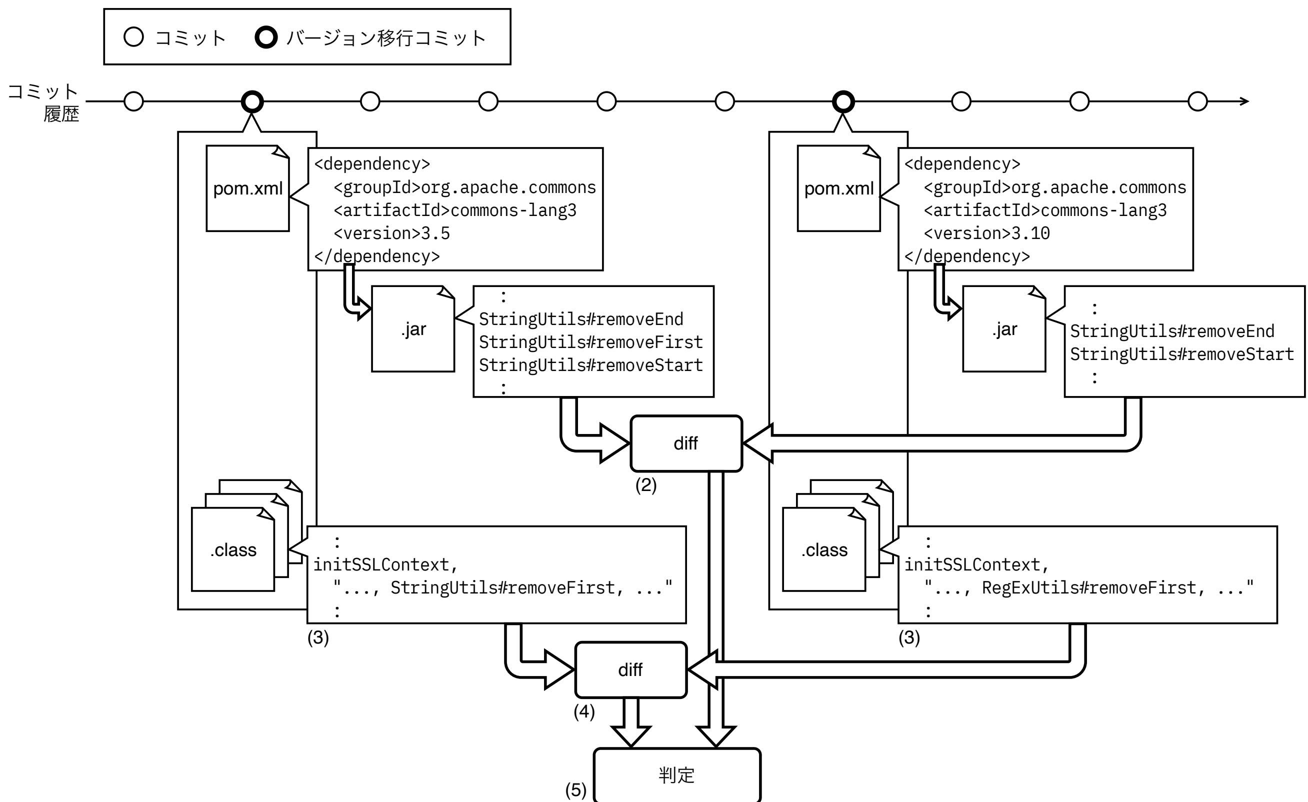

図 1 提案手法の概略

```
1 <dependency>
2   <groupId>org.apache.maven</groupId>
3   <artifactId>maven-model</artifactId>
4   <version>3.9.8</version>
5 </dependency>
```

図 2 dependency 要素の例

3.6 実装と制限

3.6.1 pom.xml

POM 変更コミットは `git-log` コマンドで`--first-parent` オプションを有効にして取得する．pom.xml から利用ライブラリの情報を抽出するために，pom.xml を読み込んでアクセス API を提供するライブラリ `org.apache.maven:maven-model` を利用する．pom.xml の読み込み時に pom.xml として正しい XML 文書であるか検査できるため，pom.xml が不正なコミットはスキップする．

利用ライブラリのバージョンは `dependency/version` 要素で指定するが，その記述方法は柔軟性が高く，バージョンを一意に指定できるだけでなく，許容するバージョンの範囲を指定することもできる．今回の実装では範囲指定にはほぼ対応できておらず，バージョン移行において旧バージョンが範囲指定の場合はスキップし，新バージョンが範囲指定の場合は処理時の最新バージョンとみなすようになっている．バージョンの記述に含まれる`${maven-model.version}` のようなプロパティ参照は具体値に解決する．

現在の実装は，Maven のマルチモジュールプロジェクトにも対応していない．マルチモジュールプロジェクトは，1 つの親プロジェクトの下に子プロジェクトとして複数の Maven プロジェクトをまとめたプロジェクトである．親プロジェクトと子プロジェクトのいずれにも pom.xml が存在するが，親プロジェクトの pom.xml だけを追跡する実装のため，子プロジェクトにおける利用ライブラリのバージョン移行を把握できていない．

3.6.2 JAR ファイル

利用ライブラリの JAR ファイルは Maven セントラルリポジトリからダウンロードできることを前提とする．サードパーティリポジトリで公開されているライブラリが利用される場合，pom.xml にサードパーティリポジトリの情報が追加されるため JAR ファイルの入手は不可能ではない．しかし，ライブラリとリポジトリの対応は明示されず，サードパーティリポジトリで公開されるライブラリは少数であるため，今回の実装では対応していない．

3.6.3 ビルド

手順 (3) において，プロジェクト内の各メソッド定義の中で呼び出されているメソッドのリストは，Java のソースコードではなくクラスファイルから抽出する．呼び出されているメソッドなどの完全修飾名を得る手間を省くためであり，抽出には既存のツール [15] を利用している．クラスファイルを得るためにプロジェクトをビルドする必要があるが，Maven を利用していても様々な理由でビルドできな

いプロジェクトないしはコミットが存在する．

長期間にわたって開発が継続しているプロジェクトでは，ビルドに利用する JDK のバージョン移行が行われていることがある．この場合，古いコミットから新しいコミットまで単一のバージョンの JDK のみでビルドすることは不可能である．そのため，LTS である Java 8, 11, 17, 21 および最新の Java 23 を利用し，成功するまで新しい方から順に切り替えながらビルドを行うようにしている．

Maven の利点の一つは，利用ライブラリとその依存先ライブラリを自動的にダウンロードしてクラスパスを設定することである．しかし，プロジェクトによっては必要なライブラリが入手できずビルドできないコミットが存在する．ライブラリの指定された特定のバージョンが存在しなくなった（例えば，安定版ではない `SNAPSHOT` リリースが指定されているが，その後安定版が公開されて `SNAPSHOT` リリースは削除された）ことや，ライブラリを公開していたサードパーティリポジトリ自体が消滅したことが理由である．これらは解決する手段がないため，該当のコミットはスキップせざるを得ない．

ビルドは単純に `mvn compile` コマンドによって行う．ビルドの失敗は，Java ソースコードがコンパイルできないことによるものが大半である．一方で，プロジェクトに固有の設定に起因する失敗が少数ではあるが存在する．例えば，ビルド時に行う PMD による静的検査の失敗がそのままビルドの失敗になるプロジェクトがあり，PMD を実行しないようにするといった対応が必要となるためビルドを諦めている．ソースファイルにライセンス情報が含まれていることをビルド時に検査する `license-maven-plugin` が失敗するプロジェクトや，今回の実装ではソースコードを `git-export` によって git の管理外に取り出してからビルドすることが原因で `git-commit-id-plugin` が失敗するプロジェクトもある．ビルドに失敗する原因はプロジェクトによって異なり，対応可能な場合でも個別対応が必要であるため，網羅的収集に向けた自動化を難しくする要因の一つとなっている．

4. 予備実験

GitHub で公開されているプロジェクトを対象として，提案手法が適用できることを確認する．実装の都合上，手順 (6) のソースコードの差分の生成は省略したが，手順 (5) までで対象のプロジェクト，コミット ID，修正されたメソッドの名前，削除された非互換 API の名前が判明しているので，差分の生成は可能である．

4.1 対象プロジェクト

GitHub で公開されておりルートディレクトリに pom.xml が存在する Java プロジェクトの中から，2024 年 11 月 20 日時点でスター数の多い順に 91 個のプロジェクトを対象とする．GitHub におけるスター数は，プロジェクトの人気度を表す指標として一般的でかつ容易に利用できることから，スター数の上位から対象を選択している．

提案手法は，非互換 API 呼び出しの修正の実例を実プロジェクトから収集するため，適用対象のプロジェクトが多いほど，様々なライブラリの非互換 API について幅広く収集できる．GitHub で公開されているスター数が 100 個以上の Java プロジェクトは約 24,000 個存在する．そのうち，構成管理に Maven を利用しているプロジェクトが約 7,800 個，Gradle を利用しているプロジェクトが約 10,000 個であり，その他はいずれも利用していないプロジェクトである（ただし，両方を利用しているように見えるプロジェクトがごく少数存在する）．なお，プロジェクトのルートディレクトリに pom.xml が存在すれば Maven を，gradlew あるいは gradlew.bat が存在すれば Gradle を利用していると判断する．Maven と同様に Gradle も Java プロジェクトの構成管理の主流ツールであり，手順 (1) を Gradle 向けに実現して収集範囲を広げる必要がある．

4.2 適用結果

全部で 91 個のプロジェクトのうち 21 個のプロジェクトから非互換 API 呼び出しの修正を 559 件収集できた．

すべてのプロジェクトにおいて 2 件以上の POM 変更コミットが存在し，全体では 29,437 件の POM 変更コミットが存在する．82 個のプロジェクトで pom.xml に `dependency` 要素が存在し，Java 標準ライブラリ以外のライブラリが利用されている．全体で利用されているライブラリは 2,868 種類である．70 個のプロジェクトで利用ライブラリのバージョン移行が 1 回以上行われており，全体では 6,059 個のバージョン移行コミット，52,696 件のバージョン移行が存在する[*1]．プロジェクトのビルドは，全体を通してバージョン移行コミットとその親コミットの計 11,030 個のコミットに対して行われたが，成功は 2,393 件であり 8 割近くが失敗している．

修正を収集できた 21 プロジェクトの概要を表 1 に示す．収集した修正数は，同じ修正内容でも異なるメソッド定義に出現していれば別にカウントしている．修正内容が同一とは限らないが，削除された非互換 API の呼び出しが同じものを 1 件と数えると，全部で 199 件である．各プロジェクトの開発期間（初回コミットから最新コミットまでの期間）は 5 年弱から 13 年程度で，大半のプロジェクトで直近 1 年以内にコミットが行われている．大まかな傾向として，開発期間が長く利用ライブラリ数が多いと POM 変更コミット数が多く，バージョン移行数は利用ライブラリ数と正の相関がある．一方で，利用している API の互換性が維持され続けていれば API 呼び出しの修正は発生しないた

[*1] ただし，同一ライブラリの同一バージョン間の移行でもプロジェクトが異なれば別にカウントしている．

表 1　非互換 API 呼び出しの修正を収集した 21 プロジェクトの概要

プロジェクト	スター数	最新コミット日	開発期間（月数）	POM 変更コミット数	利用ライブラリ数	バージョン移行数	収集した修正数
@xkcoding/spring-boot-demo	33,137	2022/09/02	58	71	4	20	1
@alibaba/easyexcel	32,688	2024/10/29	80	85	20	65	20
@alibaba/nacos	30,340	2024/11/22	76	235	103	1,325	39
@binarywang/WxJava	30,004	2024/11/24	123	234	23	38	1
@alibaba/canal	28,531	2024/10/14	121	141	56	104	12
@YunaiV/ruoyi-vue-pro	27,888	2024/11/25	61	137	57	48	1
@crossoverJie/JCSprout	27,077	2024/05/21	77	17	19	4	1
@apache/flink	24,134	2024/11/30	167	910	112	301	218
@keycloak/keycloak	23,517	2024/12/06	137	878	506	8,330	111
@elunez/eladmin	21,345	2024/09/19	69	57	40	21	1
@brettwooldridge/HikariCP	20,023	2024/11/30	133	387	13	54	2
@linlinjava/litemall	19,283	2024/06/16	75	40	29	26	6
@infinilabs/analysis-ik	16,612	2024/12/12	156	96	5	79	25
@Konloch/bytecode-viewer	14,706	2024/10/18	120	89	103	250	1
@apache/druid	13,520	2024/12/20	146	1,439	214	915	12
@apache/dolphinscheduler	12,881	2025/01/02	70	328	197	302	2
@pagehelper/Mybatis-PageHelper	12,221	2024/09/30	123	144	14	27	14
@macrozheng/mall-swarm	11,882	2025/01/13	81	29	35	34	7
@cryptomator/cryptomator	11,853	2025/01/12	131	209	26	276	81
@code4craft/webmagic	11,438	2024/12/31	140	190	31	79	2
@debezium/debezium	10,696	2025/01/15	110	712	62	1,341	2

め，バージョン移行数が多いとしても非互換 API 呼び出しの修正も多いとは限らない．

4.3　収集された修正の実例

`@binarywang/WxJava` プロジェクトを例として，非互換 API 呼び出しの修正の収集までの過程を示す．なお，実験時の最新のコミットは d2ff40f5 であり，主ブランチは `develop` である．

POM 変更コミットは全部で 234 件，そのうちバージョン移行コミットが 29 件である．表 2 に示すように全体で 14 種類のライブラリについて 38 件のバージョン移行が行われており，複数のメソッドが各バージョン移行における新旧バージョン間で削除されている．なお，表 2 では古いコミットから新しいコミットの順に並んでいる．

これらのバージョン移行の中で，削除されたメソッドに対応する修正はコミット 9ab2cfb2 のみで行われているので，このコミット（以下，移行コミットと呼ぶ）に着目する．表 2 より，移行コミットではライブラリ `org.apache.commons:commons-lang3` のバージョンが 3.5 から 3.10 に更新され，ライブラリ内の 53 個のメソッドが削除されている．一方，移行コミットにおいてプロジェクト内で定義されているメソッドは 15,379 個で，移行コミットの親コミットでは 13,819 個である．両コミットで定義されているメソッドのうち 578 個のメソッドにおいて，本体内で呼び出されているメソッド群が親コミットと移行コミットで異なる．この 578 個のメソッドの中で，親コミット側で呼び出され移行コミット側で呼び出されないメソッド群（すなわち，呼び出されなくなったメソッド群）の中にライブラリで削除された 53 個のメソッドのいずれかを含むメソッドは `com.github.binarywang.wxpay.config.WxPayConfig#initSSLContext` のみであり，呼び出されなくなったメソッドは `org.apache.commons.lang3.StringUtils#removeFirst` である．以上から，移行コミットにおける `initSSLContext` の修正に，非互換 API 呼び出しの修正が含まれていると判断する．

コミット履歴を調べると，移行コミットにおける diff の中に図 3 に示す内容が含まれている．この内容から `org.apache.commons.lang3.StringUtils#removeFirst` の代わりに `org.apache.commons.lang3.RegExUtils#removeFirst` を利用するように修正されたことがわかる．

なお，バージョン 3.10 の `org.apache.commons.lang3.StringUtils#removeFirst` のドキュメンテーションコメントの中に "@deprecated Moved to RegExUtils." と書かれており，生成された Javadoc にも同じ内容が含まれているため，この例では Javadoc を見ればどのように修正すればよいかヒントが得られる．一方，この例のように `@Deprecated` アノテーションを付けるのではなくファイルから削除されたメソッドは Javadoc には含まれないため，移行先メソッドや代替手段を何らかの文書に残す必要があり，このような文書が存在しないケースに対して提案手法は有効である．

表 2 WxJava プロジェクトにおけるバージョン移行

バージョン移行コミット	ライブラリ	旧バージョン	新バージョン	削除メソッド数
acedf395	org.apache.httpcomponents:fluent-hc	4.3.5	4.5	4
acedf395	org.apache.httpcomponents:httpmime	4.3.5	4.5	12
331f06ef	com.google.code.gson:gson	2.2.2	2.7	108
331f06ef	commons-codec:commons-codec	1.9	1.10	0
331f06ef	commons-io:commons-io	2.4	2.5	1
1f15cc82	org.apache.commons:commons-lang3	3.4	3.5	34
491095f1	com.google.code.gson:gson	2.7	2.8.0	0
491095f1	com.google.guava:guava	19.0	20.0	428
491095f1	org.slf4j:slf4j-api	1.7.10	1.7.24	1
cc8359b2	com.thoughtworks.xstream:xstream	1.4.9	1.4.10	6
4223abfa	com.thoughtworks.xstream:xstream	1.4.10	1.4.11	11
7bd4726e	com.thoughtworks.xstream:xstream	1.4.11	1.4.11.1	0
9ab2cfb2	org.apache.commons:commons-lang3	3.5	3.10	53
2769f506	com.google.guava:guava	20.0	29.0-android	755
c01347ca	com.google.guava:guava	29.0-android	29.0-jre	80
07352d18	org.slf4j:slf4j-api	1.7.24	1.7.30	0
940f69a7	com.thoughtworks.xstream:xstream	1.4.11.1	1.4.14	5
573bd450	com.thoughtworks.xstream:xstream	1.4.14	1.4.15	0
84701228	com.thoughtworks.xstream:xstream	1.4.15	1.4.16	0
51dc6619	commons-io:commons-io	2.5	2.7	11
f64addb0	com.thoughtworks.xstream:xstream	1.4.16	1.4.17	0
c7c834b4	com.google.guava:guava	29.0-jre	30.0-jre	42
3ab66e34	org.apache.httpcomponents:httpclient	4.5	4.5.13	14
3ab66e34	org.apache.httpcomponents:httpmime	4.5	4.5.13	0
159471c9	com.thoughtworks.xstream:xstream	1.4.17	1.4.18	1
2914943c	com.thoughtworks.xstream:xstream	1.4.18	1.4.19	0
9517292a	com.google.code.gson:gson	2.8.0	2.8.9	79
4e6e692d	commons-codec:commons-codec	1.10	1.13	39
4ee77280	com.thoughtworks.xstream:xstream	1.4.19	1.4.20	0
0b4f2c41	org.bouncycastle:bcpkix-jdk15on	1.68	1.70	19
b1015a35	com.google.guava:guava	30.0-jre	32.0.0-jre	225
e61cf93a	com.fasterxml.jackson.dataformat:jackson-dataformat-xml	2.13.0	2.15.2	4
e61cf93a	com.github.binarywang:qrcode-utils	1.1	1.3	1
e61cf93a	com.google.code.gson:gson	2.8.9	2.10.1	77
e61cf93a	com.google.guava:guava	32.0.0-jre	32.1.2-jre	3
01f8e819	org.bouncycastle:bcpkix-jdk18on	1.77	1.78	12
11d525c5	org.bouncycastle:bcpkix-jdk18on	1.78	1.78.1	0
172a49c1	commons-io:commons-io	2.7	2.14.0	20

```
1 @@ -138,16 +186,20 @@ public class WxPayConfig {
2 ...snip...
3       if (this.getKeyPath().startsWith(prefix)) {
4 -         String path = StringUtils.removeFirst(this.getKeyPath(), prefix);
5 +         String path = RegExUtils.removeFirst(this.getKeyPath(), prefix);
6           if (!path.startsWith("/")) {
```

図 3 移行コミット 9ab2cfb2 における diff の一部

5. おわりに

本稿では，利用ライブラリのバージョン移行を支援するために，ライブラリが提供する API の中で更新時に互換性が維持されなくなった API を利用するソースコードの修正を網羅的に収集する手法を提案した．GitHub で公開されている 91 個のプロジェクトに提案手法を適用する予備実験を行い，意図するソースコード修正が収集できることを確認した．

今後の課題として，適用対象のプロジェクトを拡大すること，Maven 以外の構成管理ツールに対応すること，Java 以外のプログラミング言語に対応すること，該当箇所の

© 2025 Information Processing Society of Japan

ソースコード差分を抽出した上で，ライブラリ名，バージョン番号，API 名などと組み合わせてデータベース化することなどが挙げられる．

謝辞 本研究の一部は，JSPS 科研費 JP24K14906 および 2024 年度南山大学パッヘ研究奨励金 I-A-2 の助成による．

参考文献

[1] Raemaekers, S., van Deursen, A. and Visser, J.: Measuring Software Library Stability through Historical Version Analysis, *Proceedings of the 28th IEEE International Conference on Software Maintenance*, pp. 378–387 (2012).

[2] Cadariu, M., Bouwers, E., Visser, J. and van Deursen, A.: 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] McIntosh, S., Adams, B., Nguyen, T. H., Kamei, Y. and Hassan, A. E.: An Empirical Study of Build Maintenance Effort, *Proceedings of the 33rd International Conference on Software Engineering*, Association for Computing Machinery, pp. 141–150 (2011).

[4] McDonnell, T., Ray, B. and Kim, M.: An Empirical Study of API Stability and Adoption in the Android Ecosystem, *2013 IEEE International Conference on Software Maintenance*, pp. 70–79 (2013).

[5] Kula, R. G., German, D. M., Ouni, A., Ishio, T. and Inoue, K.: Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration, *Empirical Software Engineering*, Vol. 23, No. 1, pp. 384–417 (online), DOI: 10.1007/s10664-017-9521-5 (2018).

[6] Mostafa, S., Rodriguez, R. and Wang, X.: A Study on Behavioral Backward Incompatibilities of Java Software Libraries, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, pp. 215–225 (2017).

[7] Xavier, L., Brito, A., Hora, A. and Valente, M. T.: 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] 渥美紀寿，桑原寛明：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，信学技報，SS2018-78, Vol. 118, No. 471, pp. 157–162 (2019).

[9] Fujibayashi, D., Ihara, A., Suwa, H., Kula, R. G. and Matsumoto, K.: Does the Release Cycle of a Library Project Influence When It Is Adopted by a Client Project?, *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 569–570 (online), DOI: 10.1109/SANER.2017.7884681 (2017).

[10] Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y. and Liu, Y.: An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects, *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 35–45 (online), DOI: 10.1109/ICSME46990.2020.00014 (2020).

[11] Alrubaye, H., Mkaouer, M. W. and Ouni, A.: MigrationMiner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level, *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 414–417 (online), DOI: 10.1109/ICSME.2019.00072 (2019).

[12] Teyton, C., Falleri, J.-R. and Blanc, X.: Automatic Discovery of Function Mappings between Similar Libraries, *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 192–201 (online), DOI: 10.1109/WCRE.2013.6671294 (2013).

[13] He, H., Xu, Y., Ma, Y., Xu, Y., Liang, G. and Zhou, M.: A Multi-Metric Ranking Approach for Library Migration Recommendations, *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 72–83 (online), DOI: 10.1109/SANER50967.2021.00016 (2021).

[14] Huang, K., Chen, B., Pan, L., Wu, S. and Peng, X.: RepFinder: Finding Replacements for Missing APIs in Library Update, *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pp. 266–278 (online), DOI: 10.1109/ASE51524.2021.9678905 (2021).

[15] 桑原寛明，渥美紀寿：API 利用パターンを用いた自動プログラム修正におけるパターン検索の予備評価，情報処理学会研究報告ソフトウェア工学，Vol. 2022-SE-210, No. 29, pp. 1–8 (2022).

© 2025 Information Processing Society of Japan