利用ライブラリの進化に追随するためのソースコード修正例の検索手法

桑原 寛明　渥美 紀寿

本稿では，ソフトウェア開発において利用ライブラリのバージョン移行を行う際に，削除されるメソッドの呼び出しの修正を支援する手法を提案する．ライブラリの進化に伴って API が変更されると，API を利用するソースコードの修正が必要となるが，修正方法が必ずしも文書化されているとは限らない．提案手法では，既存のプロジェクトから収集された，利用ライブラリのバージョン移行と同時に実施された被削除メソッドの呼び出しの修正のデータを検索し，実施したい利用ライブラリのバージョン移行に必要なソースコード修正を提示する．実プロジェクトで実施された利用ライブラリのバージョン移行を対象とする適用実験の結果を示す．

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して効率的に開発を進めることが一般的である．特に，プログラムの作成時に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．様々な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 [7].

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いる．更新には新機能の追加の他に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バグや脆弱性が残り続ける．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る [5]．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，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 [3].

ライブラリの進化により，提供される API の後方互換性が維持されなくなることがある．API の変更の約 15%が後方互換性を維持していないこと [10] や，ライブラリのドキュメント類に互換性に関する情報が記載されることは少ないこと [6] も報告されている．ライブラリの進化に伴って互換性が維持されなくなった 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発生する．

本稿では，API の後方互換性が維持されず，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．同じバージョン移行を既に行ったソフトウェアにおいては，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．提案手法では，開発中のソフトウェアでバージョン移行するライブラリと新旧のバージョンを入力とし，新旧

A Method for Retrieving Source Code Modifications for Adopting Library Evolution

Hiroaki Kuwabara, 南山大学 理工学部, Faculty of Science and Technology, Nanzan University.

Noritoshi Atsumi, 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.

バージョン間で削除された API のうち開発中のソフトウェアが利用している API を特定する．特定された API の呼び出しをバージョン移行に伴って修正している OSS プロジェクトのコミットを検索する．

先行研究[4] で，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，特にライブラリの更新に伴って削除された API の呼び出しに対する修正を OSS プロジェクトから網羅的に収集する手法を提案した．データベースの構築に向けて収集を進めており，本研究ではこの結果を活用する．

2 ソースコード修正の収集

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 を利用していた場合，当該 API を利用せず機能を維持するためにソースコードも同時に修正される可能性が高い．そのため，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，非互換 API を呼び出すコード周辺の修正を抽出できる．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れば，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．

先行研究[4] では，ソースコードなどのファイルの変更を Git で管理しており，利用ライブラリなどの構成管理に Maven を利用している Java プロジェクトを対象として，ライブラリの更新に伴って削除されたメソッド（以下，非互換メソッドと呼ぶ）を呼び出すコードに対する修正を抽出する手法を実装している．抽出の手順は以下の通りである．

1. 利用ライブラリのバージョン移行が行われた主ブランチ上のコミット（以下，バージョン移行コミットと呼ぶ），各バージョン移行コミットにおける移行対象のライブラリ，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主ブランチ上の親コミットそれぞれにおいて，ソースコード中のメソッド定義毎に，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けるメソッド定義毎に，3. で抽出されたリストを親コミットにおける同一のメソッド定義と比較し，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，バージョン移行コミットで非互換メソッドの呼び出しが修正されたと判断する．

抽出結果として，以下に示す情報の組を 1 つのエントリとして記録する．

- プロジェクト名（GitHub 上のリポジトリ名）
- リポジトリ所有者のアカウント名
- ライブラリ名（`groupId` と `artifactId` のペア）
- 移行前後のライブラリのバージョン
- バージョン移行コミットとその親コミットのコミット ID
- バージョン移行コミットで非互換メソッドの呼び出しが削除されたプロジェクト内のメソッド名
- このメソッド内で呼び出しが削除された非互換メソッドの集合

バージョン移行コミットとその親コミットがわかるため，修正によるソースコードの差分を取得できる．

3 ソースコード修正の検索手法

3.1 手順

開発中のプロジェクトにおいて，利用ライブラリのバージョン移行に伴ってソースコードの修正が必要となる場合，必要な修正の具体的な内容を把握したい．ライブラリの開発者が文書やサンプルコードを提供していればそれを参照すればよいが，提供していない場合は別の手段が必要である．

開発者は移行対象の利用ライブラリとその移行前後のバージョンを知っている．これらの情報と，様々なプロジェクトから収集されたソースコード修正のデータ（以下，収集データと呼ぶ）を利用すれば，参考になる修正がどのプロジェクトのどのコミットのどのメソッド定義内に存在するか特定できる．特定できたソースコード修正を開発者に提示すればよい．

利用ライブラリのバージョン移行に伴い必要となるソースコード修正を以下の手順で検索して提示する．

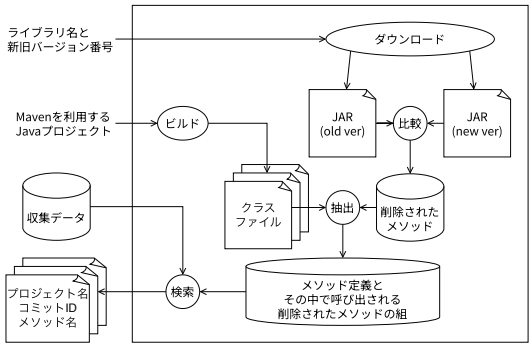

図 1　手法の概略

1. 利用ライブラリの移行前後のバージョン間で削除された非互換メソッドを抽出する．
2. 1. を利用し，プロジェクト内のメソッド定義毎に呼び出している非互換メソッドを特定する．
3. 収集データから，2. で特定された非互換メソッドの呼び出しの削除を含むエントリを検索する．
4. 検索されたエントリに基づき，該当するソースコード差分の GitHub 上のページを表示する．

3.2　非互換メソッドの抽出

ソースコード修正の収集 [4] と同じ方法で非互換メソッドを抽出する．Maven の場合，利用ライブラリは `groupId` と `artifactId` の組で一意に特定できる．利用ライブラリの移行前後のバージョン両方の JAR ファイルをダウンロードし，それぞれからライブラリ内で定義されるメソッドのシグネチャを抽出する．移行前バージョンに存在し移行後バージョンに存在しないメソッドを非互換メソッドとする．

抽出するメソッドは，API として外部に公開される `public` あるいは `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は削除されたメソッドとして扱う．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．オーバーロードされたメソッドを区別するために引数の型を含む．

3.3　非互換メソッド呼び出しの特定と提示

ソースコードの修正が必要な箇所を開発者に示すために，プロジェクト内のメソッド定義毎に，その中で呼び出されている非互換メソッドを特定する．呼び出されているすべてのメソッドについてシグネチャを抽出して非互換メソッドのシグネチャと比較する．

特定された非互換メソッドの呼び出しを削除するソースコード修正に関する情報を収集データから検索する．収集データには，既存プロジェクトで実施された利用ライブラリのバージョン移行について，どのメソッド定義内のどの非互換メソッドの呼び出しを削除したか記録されている．特定された非互換メソッドの呼び出しを削除しているエントリを探せばよい．

見つかったエントリには，その情報源である GitHub 上のリポジトリ名と所有者のアカウント名，バージョン移行コミットとその親コミットのコミット ID が含まれている．これらの情報を用いて，この 2 コミット間の差分を表示する GitHub 上のページを表示する．表示するページの URL は以下のように生成できる．

```
https://github.com/<owner>/<repository>
/compare/<from>..<to>?diff=split&w=1
```

ここで，`<owner>`は所有者のアカウント名，`<repository>`はリポジトリ名であり，`<to>`と`<from>`はそれぞれバージョン移行コミットと親コミットのコミット ID である．`diff=split` は差分を左右に並べて（side by side で）表示すること，`w=1` は空白に関する差分を表示しないことの指定である．

4　適用実験

4.1　概要

提案手法を実装して GitHub で公開されているプロジェクトに対して適用し，利用ライブラリのバージョン移行に伴って必要となるソースコードの修正例を検索する実験を行う．実プロジェクトにおいて実施されたバージョン移行の対象ライブラリと移行前後のバージョンを指定して，非互換メソッドの呼び出しに対する修正例を検索できるか確認する．

4.2　ソースコード修正の収集データ

実験では収集データとして [4] で収集したデータを利用する．本データの収集元は，GitHub で公開されており，ルートディレクトリに pom.xml が存在することから Maven を利用していると判断できる Java プ

ロジェクトのうち，2024 年 11 月 20 日時点のスター数が多い順に 91 個のプロジェクトである．

91 個のうち 21 個のプロジェクトにおいて，利用ライブラリのバージョン移行と同時に非互換メソッドの呼び出しを削除するコミットが存在しており，収集された非互換メソッドの呼び出しの削除を含む修正は全部で 559 件である．なお，削除対象が同じでも異なるメソッド定義に出現していれば個別にカウントされている．559 件の収集データに含まれるライブラリは 38 種類であり，バージョン移行に伴って呼び出しが削除された非互換メソッドは全部で 263 種類である．

4.3 適用対象プロジェクト

収集データの収集元ではないプロジェクトを適用対象とする．具体的には，GitHub で公開されており，収集元プロジェクトと同様に Maven を利用していると判断できる Java プロジェクトのうち，2024 年 11 月 20 日時点のスター数が多い順に 101 から 200 番目の 100 個のプロジェクトに適用する．

4.4 実験手順

各対象プロジェクトに対して，以下の手順で進める．

1. ソースコード修正の収集手法と同様にして，バージョン移行コミットを列挙し，各バージョン移行コミットについて，その親コミット，移行対象のライブラリと移行前後のバージョンを特定する．
2. 移行対象のライブラリが収集データに含まれるものを抽出する．
3. 移行前後のバージョンについて，メジャーバージョンあるいはマイナーバージョンが更新されるものを抽出する．
4. バージョン移行コミットを選択し，移行対象のライブラリの移行前後のバージョン間で削除される非互換メソッドの中で，親コミット内で呼び出されているメソッドを特定する．
5. 収集データを検索し，呼び出しが削除された非互換メソッドの集合に 4. で特定された非互換メソッドを含むエントリを特定する．
6. 特定されたエントリごとに，バージョン移行コミットと親コミット間の差分を表示する GitHub 上のページの URL を生成する．

収集データに含まれていないライブラリについては検索する必要がないため，手順 2. で除外している．

多くのライブラリでセマンティックバージョニングが採用されている．セマンティックバージョニングにおけるバージョン番号は 3 つの非負整数からなる `X.Y.Z` の形式であり，`X` がメジャーバージョン，`Y` がマイナーバージョン，`Z` がパッチバージョンを表す．API の互換性を維持しない変更を行う場合はメジャーバージョンを上げるルールであるが，ルールに従わないライブラリも存在するため，手順 3. でメジャーバージョンが更新されたものに加えてマイナーバージョンのみが更新されたものも抽出する．

バージョン移行コミットにおいて移行前後のバージョンがわかるため，この 2 バージョン間の非互換メソッドの集合が得られる．これらの非互換メソッドの呼び出しが修正対象となるため，手順 4. で移行前である親コミットにおけるソースコードの中で呼び出されている非互換メソッドを特定する．特定された非互換メソッドの呼び出しを削除するソースコード修正を収集データ内で検索する（手順 5.）．

4.5 結果

対象の 100 個のうち 77 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，全体でバージョン移行コミットは 6,954 個，バージョン移行の総数は 33,958 件である．バージョン移行対象のライブラリは全部で 3,005 種類であるが，このうちの 22 種類が収集データに含まれており，この 22 種類のライブラリを対象とするバージョン移行は全部で 441 件である．441 件のうちメジャーバージョンあるいはマイナーバージョンを更新するバージョン移行は 213 件であり，この 213 件のバージョン移行について，プロジェクト内に存在する非互換メソッドの呼び出しに対する修正案を検索できるか確認した．

各バージョン移行について，バージョン移行コミットの親コミット内で呼び出されている非互換メソッドを特定する．メソッドの完全修飾名を Java クラスファイルから得るため，親コミットを `git checkout` してビルドする．213 件のバージョン移行の中には同

表 1　非互換メソッドの呼び出しを含む移行前コミット

プロジェクト	コミット ID	ライブラリ	移行前バージョン	移行後バージョン
@apache/shenyu	7b6d2912	com.alibaba.nacos:nacos-client	2.0.4	2.2.4
@flowable/flowable-engine	48251546	org.apache.commons:commons-lang3	3.1	3.3.2
@Graylog2/graylog2-server	cf709627	com.jayway.jsonpath:json-path	0.9.1	1.2.0
@Graylog2/graylog2-server	7f5a9eb5	org.elasticsearch:elasticsearch	1.3.7	1.4.4
@Graylog2/graylog2-server	f350542d	org.elasticsearch:elasticsearch	1.4.4	1.5.1
@Graylog2/graylog2-server	9c07a0e2	org.elasticsearch:elasticsearch	1.5.2	1.6.0
@jetlinks/jetlinks-community	9e8efece	org.elasticsearch:elasticsearch	7.9.2	7.10.2
@Nepxion/Discovery	ae273af2	com.alibaba.nacos:nacos-client	0.1.0	0.2.0

表 2　呼び出されていた非互換メソッド（抜粋）

コミット ID	メソッド
7b6d2912	com.alibaba.nacos.common.utils.DateFormatUtils#format(java.util.Date,String)
48251546	org.apache.commons.lang3.ObjectUtils#equals(Object,Object)
	org.apache.commons.lang3.ObjectUtils#toString(Object)
	org.apache.commons.lang3.ObjectUtils#toString(Object)
ae273af2	com.alibaba.nacos.api.NacosFactory#createConfigService(java.util.Properties)
	com.alibaba.nacos.api.config.ConfigService#publishConfig(String,String,String)
	com.alibaba.nacos.api.config.ConfigService#getConfig(String,String,long)
	com.alibaba.nacos.api.config.ConfigService#removeConfig(String,String)
	com.alibaba.nacos.api.PropertyKeyConst#<init>()

一のコミットで実施されたものがあるため，対象の親コミットは全部で 205 個である．そのうち，ビルドに成功したコミットが 62 個，Maven マルチモジュール構成の一部のモジュールがビルドできたコミットが 60 個，ビルドに失敗したコミットが 83 個であった．

ビルドで得られた Java クラスファイルの解析結果から，非互換メソッドの呼び出しを含む親コミットが 5 個のプロジェクトに合計 8 件存在した．8 件の内容を表 1 に示す．呼び出されていた非互換メソッドは全部で 19 種類であり，一部を表 2 に示す．各非互換メソッドについて，その呼び出しの削除を含むエントリは収集データに存在しておらず，今回の実験では，バージョン移行に伴って必要となるソースコードの修正例を見つけられなかった．

4.6　考察

今回の実験の範囲内では，利用ライブラリのバージョン移行に伴うソースコード修正を検索できなかった．適用対象プロジェクトにおけるバージョン移行と同時に修正された非互換メソッドの呼び出しに対して，同じメソッドの呼び出しに対する修正が収集データの収集元プロジェクトに出現していないことが直接の原因である．

実験で用いた収集データは，エントリが 559 件で，含まれるライブラリが 38 種類，非互換メソッドが 263 種類と非常に小規模であり，ライブラリの種類も非互換メソッドの種類もまったく不足している．その原因として，収集対象のプロジェクトが 91 個と少ないこと，収集時に実行されるバージョン移行コミットとその親コミットのビルドの 8 割近くが失敗していること [4] が挙げられる．実験で用いた適用対象プロジェクトでも，マルチモジュール構成の一部モジュールのビルド失敗も含めれば，全体のおよそ 7 割でビルドに失敗している．これらの問題を解決して，収集データの規模を大きくした上で評価実験を行う必要がある．

5 関連研究

Fujibayashi ら[1] は，23 のライブラリのリリースサイクルと 415 の Apache Software Foundation の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調査し，リリースサイクルが短いライブラリの方が早いタイミングでバージョン移行が行われることを明らかにした．Wang ら[9] は，806 の OSS プロジェクト，13,565 のライブラリ，544 のセキュリティバグを用いた調査から，多くのプロジェクトで最新バージョンと隔たりの大きな古いライブラリが利用されていることを明らかにした．

Huang ら[2] は，ライブラリの更新時に削除された API に対する代替 API を更新後のライブラリのソースコードなどから自動検出する手法を提案し，RepFinder を実装した．Schäfer[8] らは，ライブラリを利用するクライアントのソースコードの変更からパターンマイニングする手法を提案した．本稿の提案手法は，[8] と同様にクライアントのソースコードから抽出するが，削除された API を一意に特定した上でその API の呼び出しの修正を検索する点が異なる．

6 おわりに

本稿では，利用ライブラリのバージョン移行を行う際に，非互換メソッドの呼び出しの修正を支援する手法を提案した．GitHub で公開されているプロジェクトから収集された，利用ライブラリのバージョン移行と同時に実施された非互換メソッドの呼び出しに対するソースコード修正のデータを検索し，実施したいバージョン移行の対象ライブラリ，移行前後のバージョン，プロジェクト内で呼び出されているメソッドに合致するソースコード修正の実例を提示する．

既存研究において収集されたソースコード修正のデータを利用し，収集元とは異なる 100 個のプロジェクトにおいて実施されたバージョン移行に対して適用実験を行った．その結果，非互換メソッドの呼び出しは存在していたが，収集データには当該メソッドの呼び出しの修正が含まれておらず，ソースコード修正の実例の提示には至らなかった．

今後の課題として，ソースコード修正の大規模な収集データを構築して再評価すること，適切なソースコード修正の箇所をピンポイントで提示する手法を構築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 2025 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] 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?, *Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 569–570.

[2] 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*, 2021, pp. 266–278.

[3] 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(2018), pp. 384–417.

[4] 桑原寛明, 渥美紀寿: 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, 情報処理学会研究報告ソフトウェア工学, Vol. 2025-SE-219, No. 25, 2025, pp. 1–8.

[5] 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*, 2011, pp. 141–150.

[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*, 2017, pp. 215–225.

[7] 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*, 2012, pp. 378–387.

[8] Schäfer, T., Jonas, J., and Mezini, M.: Mining Framework Usage Changes from Instantiation Code, *Proceedings of the 2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 471–480.

[9] 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, *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*, 2020, pp. 35–45.

[10] 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*, 2017, pp. 138–147.