Classification of Changes based on API

Masashi Iriyama[1], Yoshiki Higo[1], and Shinji Kusumoto[1]

Graduate School of Information Science and Technology, Osaka University, Japan
{m-iriyam, higo, kusumoto}@ist.osaka-u.ac.jp

Abstract. In software maintenance process, software libraries are occasionally updated, and their APIs may also be updated. API changes can be classified into two categories: changes that break backward compatibility (in short, breaking changes) and changes that maintain backward compatibility (in short, maintaining changes). Detecting API changes and determining whether each is a breaking or maintaining change is useful for code reviews and release note generations. Since it is burdensome to check API changes manually, research on automatic detection of API changes has been conducted. APIDiff is a tool that automatically detects API changes and classifies the detected changes into breaking and maintaining ones. APIDiff takes two versions of a Java library as input, and it detects API changes based on the similarity of the input code. Each detected change is classified into the two kinds of changes. However, since APIDiff identifies breaking changes for each type of change, it tends to fail to correctly classify changes if multiple changes were conducted to a single API. On the other hand, our proposed technique in this paper groups changes by APIs and checks whether each group contains changes that break backward compatibility. Classifying API changes more correctly by our technique will be helpful for release note generations in maintenance process. We conducted experiments on eight open-source software and confirmed that our technique could detect API changes more correctly than APIDiff. We also confirmed that the proposed technique could classify API changes more correctly into breaking and maintaining ones than APIDiff.

Keywords: API Evolution · Breaking Changes · Mining Software Repositories

1 Introduction

Libraries have been used in many software applications [6]. Libraries provide functionality through application programming interfaces (in short, APIs). In software maintenance process, software libraries are occasionally updated, and their APIs may also be updated; API changes may include additions of new features, removals of unnecessary features, or refactoring to improve maintainability [4]. Those changes can be categorized as those that break backward compatibility (in short, *breaking changes*) and those that maintain backward

MPChartLib/src/main/java/com/github/mikephil/charting/components/YAxis.java

```
    public class YAxis extends AxisBase {
        ...
-       public void setValueFormatter(YAxisValueFormatter f) {
-           if (f == null)
-               mYAxisValueFormatter = new DefaultYAxisValueFormatter(mDecimals);
-           else
-               mYAxisValueFormatter = f;
-       }
    }
```

MPChartLib/src/main/java/com/github/mikephil/charting/components/AxisBase.java

```
    public abstract class AxisBase extends ComponentBase {
        ...
+       public void setValueFormatter(AxisValueFormatter f) {
+           if (f == null)
+               mAxisValueFormatter = new DefaultAxisValueFormatter(mDecimals);
+           else
+               mAxisValueFormatter = f;
+       }
    }
```

https://github.com/PhilJay/MPAndroidChart/commit/1482f9331e6d47c2e255be1cb95b3e91133aabc0

Fig. 1: An example of an issue in APIDiff

compatibility (in short, *maintaining changes*). Detecting API changes and determining whether the changes maintain backward compatibility of the API is useful for code reviews and release note generations [7].

Since manually detecting API changes is burdensome, research has been conducted on automatically detecting API changes. APIDiff is a tool that automatically detects API changes and classifies them into breaking and maintaining ones [1]. A variety of research has been conducted using APIDiff. For example, research has been conducted to clarify the stability of libraries [10], the impact of breaking changes on client code [10], reasons why developers made breaking changes [2], and developers' awareness of the dangers of breaking changes [11].

However, APIDiff tends to fail to correctly classify changes if multiple changes were conducted to a single API since it identifies breaking changes for each type of change. As a result, API developers (library developers) and API users (library users) may have wrong perceptions of API changes. Fig. 1 shows an example of the issue in APIDiff. APIDiff should classify the API changes of `setValueFormatter` into *Pull Up Method* and *Change in Parameter List*. Users of `setValueFormatter` can no longer use it after the API changes because the parameter of the API has been changed. That is, the backward compatibility of `setValueFormatter` is broken by the changes, but APIDiff classifies *Pull Up Method* incorrectly into the maintaining change based on its change type.

Our proposed technique groups changes by APIs and checks whether each group contains API changes that break backward compatibility. Classifying API changes more correctly by our technique will be helpful for release note generations in maintenance process. We conducted experiments on eight open-source software and confirmed that our technique could detect API changes more correctly than APIDiff. We also confirmed that our technique could classify API changes more correctly into breaking and maintaining ones than APIDiff.

2 Preliminaries

2.1 Catalog of API changes

The backward compatibility considered in this paper is in the context of syntactic changes and not semantic changes. The catalog of breaking changes is shown in Table 1. The catalog of maintaining changes is shown in Table 2. Those catalogs are based on the `README` file of APIDiff[1] and the `README` file of RefactoringMiner[2]

2.2 APIDiff

APIDiff internally utilizes a refactoring detection tool called RefDiff [8]. RefDiff outputs a list of refactoring operations applied to the later version of the two input versions based on the similarity of the code.

The two versions of a Java library given as input to APIDiff are passed to RefDiff, and classes, methods, and fields are extracted for each version. RefDiff obtains a list of refactoring operations applied to the later version. Then refactoring operations that are not related to APIs are discarded. APIDiff itself extracts classes, methods, and fields for each version. APIDiff matches APIs between the two versions based on the list of refactoring operations and information such as fully qualified names of classes, APIs' names, and sequences of parameters. Based on the results of the API matching and information such as API qualifiers and annotations, API changes are detected. The detected changes are classified into breaking or maintaining changes based on their change types. Then APIDiff

[1] https://github.com/aserg-ufmg/apidiff
[2] https://github.com/tsantalis/RefactoringMiner

Table 1: Catalog of Breaking Changes

Type	Rename, Move, Move and Rename, Remove, Lost Visibility, Add Final Modifier, Remove Static Modifier, Change in Supertype, Remove Supertype, Extract Type, Extract Subtype
Method	Move, Rename, Remove, Push Down, Inline, Change in Parameter list, Change in Exception List, Change in Return Type, Lost Visibility, Add Final Modifier, Remove Static Modifier, Move and Rename
Field	Remove, Move, Push Down, Change in Default Value, Change in Field Type, Lost Visibility, Add Final Modifier, Rename, Move and Rename

Table 2: Catalog of Maintaining Changes

Type	Add, Extract Supertype, Gain Visibility, Remove Final Modifier, Add Static Modifier, Add Supertype, Deprecated
Method	Pull Up, Gain Visibility, Remove Final Modifier, Add Static Modifier, Deprecated, Add, Extract
Field	Pull Up, Add, Deprecated Field, Gain Visibility, Remove Final Modifier, Extract

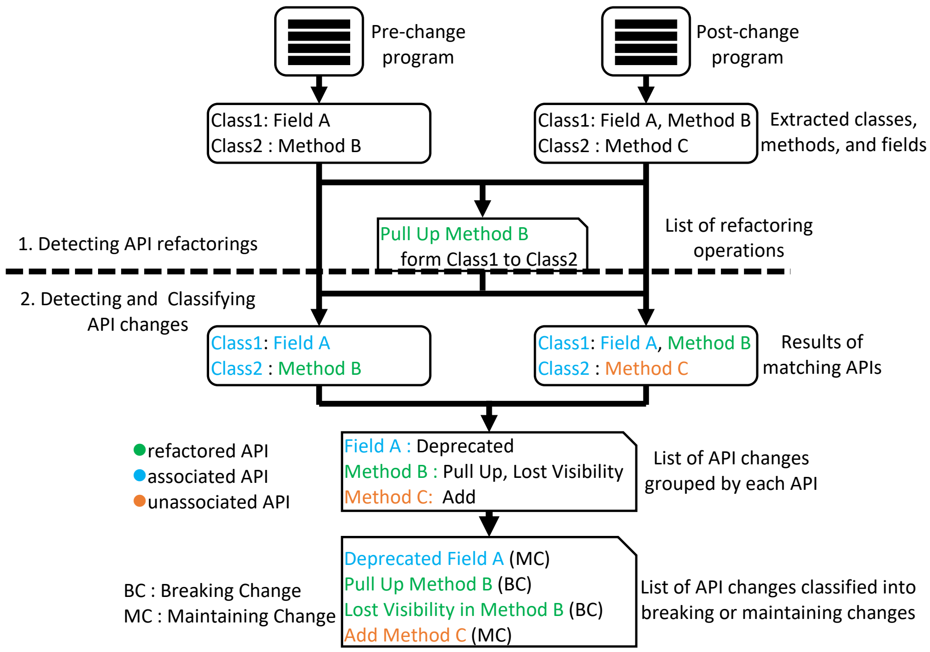

Fig. 2: Overview of the proposed technique

creates a list of API change operations, including information such as its change type, the API before and after the change, and the result of determining whether the change breaks backward compatibility.

3 Proposed Technique

An overview of our technique is shown in Fig. 2. It is important to detect API changes with high accuracy in our technique in advance to classify API changes. The proposed technique detects API refactorings using RefactoringMiner [9] instead of RefDiff. RefactoringMiner (in short, RMiner) is a tool that detects refactorings with high accuracy because of syntax-aware replacements of abstract syntax trees nodes and heuristics defined to match statements. Our proposed technique matches APIs between versions based on the output of RMiner. Our technique detects and groups changes by APIs and checks whether each group contains changes that break backward compatibility.

3.1 Detecting API refactorings

The two versions of a Java library given as input to our technique are passed to RMiner. The tool extracts classes, methods, and fields for each version. RMiner outputs a list of refactoring operations applied to the later version. Then refactoring operations that are not related to APIs are discarded.

3.2 Detecting and Classifying API changes

The API changes detection and classification procedure consists of the following steps:

Step-1 matching APIs between versions,
Step-2 detecting/grouping API changes for each API, and
Step-3 classifying API changes into breaking or maintaining changes.

In Step-1, the classes having identical fully qualified names are associated between the two versions. The methods having identical fully qualified names of the class, method names, sequences of parameters, and return types are associated. The fields having identical fully qualified names of the class, field names, and field types are associated. The unassociated APIs are classified into refactored, deleted, or added APIs based on the list of refactoring operations. In Step-2, based on the results of the API matching in Step-1 and information such as API qualifiers and annotations, API changes are detected. Then changes are grouped for each API based on the combination of the API before and after the change. In Step-3, each detected change is classified into breaking or maintaining changes based on its change type. Then our technique checks whether each group includes at least a breaking change. If the group includes at least a breaking change, our technique determines that the API is broken and reclassifies all the changes included in the group into breaking changes. Then a list of API change operations is created in the same way as APIDiff.

4 Experiment

We evaluated our technique in terms of the number of detected API changes, the precision of classifying API changes, and execution time. Our tool and datasets are available[3].

4.1 Target projects

In order to experiment with projects that are frequently updated and popular, we selected eight open source software for the experiment from the experimental targets of the longitudinal study using RMiner [3]. The eight projects were selected because their repositories included enough commits, and many users gave stars to the repositories. The target projects are shown in Table 3.

4.2 The number of detected API changes

We applied our technique and APIDiff to all the commits on the master branch of the projects and compared the number of detected changes. The results are shown in the column of Number in Table 3. While APIDiff detected 4,180 (=2,943+1,237) changes, our technique detected 7,883 (=2,943+4,940) changes. In all the projects, our technique detected more API changes than APIDiff.

[3] https://github.com/kusumotolab/APIMiner

4.3 The precision of classifying API changes

We used MPAndroidChart to calculate the precision because its calculation required manual checking of detected API changes. MPAndroidChart was also used in the experiment of APIDiff [1]. Due to the large number of API changes detected by our technique and APIDiff, we visually checked 165 API changes of which classification results are different from our technique and APIDiff. Due to the large number of API changes detected by our technique alone, 311 were sampled to achieve a tolerance of 5% and a confidence level of 95%. All the API changes detected only by APIDiff were visually checked. The results are shown in Table 4. The column of Num shows the number of detected API changes. The column of Prec1 shows whether each change is correct in change type. The column of Prec2 shows whether each change is correct in both change type and classification results. The overall precision of APIDiff alone is the number of API changes visually checked to be correct divided by 165, the number of API changes detected by APIDiff alone. The overall precision of ours alone is the number of API changes visually checked to be correct divided by the sample size, 311. Although for *Inline Method* and *Move Method*, the precision of APIDiff was higher than that of our technique, the overall precision of our technique was 89.7%, compared to 44.8% for APIDiff. The difference between Prec1 and Prec2 in the column of APIDiff alone indicates that APIDiff detected *Pull up Method* correctly but classified some of them into breaking or maintaining changes incorrectly. On the other hand, our technique detected *Pull up Method* correctly and classified them into breaking or maintaining changes correctly.

4.4 Execution time

We applied our technique and APIDiff to all the commits on the master branch of the projects and measured execution time. Then we compared the total execution time between our technique and APIDiff. The results are shown in the column of Execution Time of Table 3. In five out of the eight projects, the execution time of our technique was shorter than that of APIDiff. In three projects out of the

Table 3: Target projects, the number of detected API changes, and execution time

Project Name	LOC	Commits	Number			Execution Time			
			Both	Ours alone	APIDiff alone	Total		Detect Refactorings	
						Ours	APIDiff	Ours	APIDiff
OkHttp	72,696	4,839	675	460	396	11min53s	12min30s	11min49s	6min20s
Retrofit	26,995	1,865	243	338	84	2min36s	3min07s	2min35s	1min19s
MPAndroidChart	25,232	2,068	1,120	1,607	116	2min08s	4min18s	1min59s	2min00s
LeakCanary	26,269	1,609	41	79	51	24s	2min59s	24s	18s
Hystrix	50,510	2,108	292	722	183	19min58s	4min36s	19min56s	2min10s
iosched	23,550	2,757	91	143	44	3h16min39s	6min57s	3h16min38s	1min53s
Fresco	7,194	2,897	452	1,514	359	2min25s	20min18s	2min18s	5min46s
Logger	1,441	144	29	77	3	11s	8s	11s	3s
Sum			2,943	4,940	1,237				

eight projects, our technique took less time to detect API changes than APIDiff, even though RMiner took more time to detect refactorings than RefDiff.

5 Discussion

Fig. 3 shows an example of API change detected by APIDiff alone. APIDiff detected and classified the API change of `cloneEntry` into *Rename Method* correctly, but our technique classified the API change into *Remove Method* and *Add Method* incorrectly. Our technique matches APIs between two versions using the output of RMiner. RMiner did not detect the change, so our technique

Table 4: The precision of classifying API changes

API change type	Both				Ours alone			APIDiff alone		
	Num	Prec1	Ours Prec2	APIDiff Prec2	Num	Prec1	Prec2	Num	Prec1	Prec2
Change in Field Default Value	107				18	100	100	1	100	100
Change in Return Type Method	125				56	100	100	3	100	100
Extract Method	0				133	78.6	78.6	4	25.0	25.0
Inline Method	5				19	84.6	76.8	4	100	100
Lost Visibility in Method	19				32	38.5	38.5	44	0.0	0.0
Pull Up Method	115	100	100	0.0	107	100	100	20	100	70.0
Push Down Field	6				2	100	100	1	100	100
Push Down Method	28				29	100	100	2	100	100
Move Field	45				75	100	100	1	100	100
Move Method	60				46	15.4	15.4	12	66.7	66.7
Rename Method	147				67	100	100	22	68.2	68.2
Rename Type	27				2	100	100	2	100	100
Add Static Modifier in Method	1				3	100	100	0		
Change in Field Type	53				10	100	100	0		
Change in Supertype	132	100	100	0.0	2	100	100	0		
Deprecated Method	6				48	100	100	0		
Deprecated Type	3				2	100	100	0		
Gain Visibility in Field	43				35	100	100	0		
Gain Visibility in Method	47				56	100	100	0		
Gain Visibility in Type	2				4	100	100	0		
Lost Visibility in Field	8				18	100	100	0		
Move and Rename Type	3				2	100	100	0		
Move Type	69				8	100	100	0		
Pull Up Field	28	100	100	0.0	45	100	100	0		
Change in Parameter List	0				626	100	100	0		
Extract Field	0				3	100	100	0		
Extract Subtype	0				2	100	100	0		
Extract Supertype	0				36	92.3	92.3	0		
Extract Type	0				25	100	100	0		
Move and Rename Field	0				4	75.0	75.0	0		
Move and Rename Method	0				32	84.6	84.6	0		
Remove Static Modifier in Method	0				2	100	100	0		
Rename Field	0				58	84.6	84.6	0		
Add Final Modifier in Field	1				0			0		
Add Supertype	28				0			0		
Remove Final Modifier in Field	5				0			0		
Remove Supertype	7				0			0		
Overall	1,120	100	100	0.0	1,607	90.0	89.7	116	50.0	44.8

Pre-change

```java
protected Entry cloneEntry() {
  Entry entry = new Entry(mVal, mXIndex);
  return entry;
}
```

Post-change

```java
public Entry copy() {
  return new Entry(mVal,mXIndex);
}
```

https://github.com/PhilJay/MPAndroidChart/commit/30e54a3aa3a7a35fcd1b33f98df471c231a8740e

Fig. 3: An example of API change detected by APIDiff alone

classified `cloneEntry` into a removed API and classified `copy` into an added API incorrectly.

In the column of Ours alone in Table 4, the precision of *Move Method* was as low as 15.4%. That is because RMiner classified some of *Pull up Method* and *Push Down Method* into *Move Method* incorrectly. Our technique determines the type of refactoring based on the output of RMiner. Even if our technique classifies an API as a refactored API correctly, the type of refactoring may not be correctly determined.

In the column of Execution Time in Table 3, our technique took a much longer time to detect API changes than APIDiff in the project iosched. The majority of our tool's execution time was spent detecting refactorings by RMiner. RMiner constructs abstract syntax trees of changed files and compares subtrees of them between two versions to detect refactorings. If many files are changed in a single commit, there will be more subtrees to compare between versions, and it will take more time to detect refactorings.

6 Threats to Validity

We considered classes, methods, and fields with the access level of `public` or `protected` as APIs. The access level may be set to `public` or `protected` for internal processing rather than for exposing as an API. If such classes, methods, and fields are excluded, the experiment results may change.

In order to calculate the precision, we visually check the detected changes. Some API changes may not have been classified correctly.

Since some API change types were not detected so much, their precisions may not have been correctly calculated.

7 Related Works

RefDiff [8] and RMiner [9] are refactoring detection tools. Those tools themselves neither detect other changes (i.e., adding or removing API, etc.) nor classify detected changes into breaking or maintaining changes.

Android applications, like libraries, are suffered from API-related compatibility issues. Li et al. proposed an automated approach named CiD for systematically modeling the lifecycle of the Android APIs and analyzing app bytecode to flag usages that can lead to potential compatibility issues [5]. Our technique is for detecting API changes of Java libraries, not Android APIs.

8 Conclusions and Future Work

We proposed a new technique to classify API changes into breaking and maintaining ones automatically. Our proposed technique groups changes by APIs and checks whether each group contains changes that break backward compatibility. Classifying API changes more correctly by our technique will be helpful for release note generations in maintenance process.

By increasing the number of OSSs to be evaluated, we are going to visually check a sufficient number of API change types that were not detected so much in this experiment. We are also going to integrate our technique with CI platforms.

Acknowledgment

This research was supported by JSPS KAKENHI Japan (JP20H04166, JP21K18302, JP21K11820, JP21H04877, JP22H03567, JP22K11985)

References

1. Brito, A., Xavier, L., Hora, A., Valente, M.T.: APIDiff: Detecting API breaking changes. In: Proc. Int. Conf. Softw. Anal., Evol., Reengineering. pp. 507–511 (2018)
2. Brito, A., Xavier, L., Hora, A., Valente, M.T.: Why and how Java developers break APIs. In: Proc. Int. Conf. Softw. Anal., Evol., Reengineering. pp. 255–265 (2018)
3. Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., Chávez, A.: Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In: Proc. Joint Meeting on Founds. Softw. Eng. pp. 465–475 (2017)
4. Dig, D., Johnson, R.: How do APIs evolve? A story of refactoring. Softw. Maint., Evol.: Res., Pract. **18**(2), 83–107 (2006)
5. Li, L., Bissyandé, T.F., Wang, H., Klein, J.: CiD: Automating the Detection of API-Related Compatibility Issues in Android Apps. In: Proc. ACM SIGSOFT Int. Symp. Softw. Testing and Analysis. p. 153–163 (2018)
6. Michail, A.: Data mining library reuse patterns in user-selected applications. In: Proc. Int. Conf. Automated Softw. Eng. pp. 24–33 (1999)
7. Moreno, L., Bavota, G., Penta, M.D., Oliveto, R., Marcus, A., Canfora, G.: ARENA: An Approach for the Automated Generation of Release Notes. IEEE Trans. Softw. Eng. **43**(2), 106–127 (2017)
8. Silva, D., Valente, M.T.: RefDiff: Detecting Refactorings in Version Histories. In: Proc. IEEE/ACM Int. Conf. Mining Software Repositories. pp. 269–279 (2017)
9. Tsantalis, N., Ketkar, A., Dig, D.: RefactoringMiner 2.0. IEEE Trans. Softw. Eng. pp. 1–21 (2020)
10. Xavier, L., Brito, A., Hora, A., Valente, M.T.: Historical and impact analysis of API breaking changes: A large-scale study. In: Proc. Int. Conf. Softw. Anal., Evol., Reengineering. pp. 138–147 (2017)
11. Xavier, L., Hora, A., Valente, M.T.: Why do we break APIs? First answers from developers. In: Proc. Int. Conf. Softw. Anal., Evol., Reengineering. pp. 392–396 (2017)