ACUA: API Change and Usage Auditor

Wei Wu, Bram Adams, Yann-Gaël Guéhéneuc, Giuliano Antoniol
DGIGL, École Polytechnique de Montréal
Montréal, Canada
Email: {wei.wu, foutse.khomh, bram.adams, yann-gael.gueheneuc, giuliano.antoniol}@polymtl.ca

***Abstract*—Frameworks are widely used in modern software development. They are accessed through their Application Programming Interfaces (APIs), which specify the contract with client programs to accomplish specific tasks. When frameworks evolve, API backward-compatibility cannot always be guaranteed, yet client programs need to upgrade to use the new releases. One common reason behind client program's upgrades is the patching of security vulnerabilities. This important maintenance activity (i.e., patching security vulnerabilities) is often delayed by non-security-related API changes when the frameworks used by client programs are not up to date. In this paper, we propose a tool ACUA to generate reports containing detailed API change and usage information by analyzing the source or binary code of both frameworks and clients programs written in Java. ACUA detects the API changes between two framework releases and classify them according to the classification proposed by Des Rivières from IBM and analyzes where and how the APIs are used in client programs. Both API change and usage types influence the upgrading cost in client programs. Developers can use the API change and usage reports generated by ACUA to better understand how the differences in the APIs between two releases of a framework affect their client programs. Based on this understanding, developers can estimate the work load of upgrades and decide when to upgrade client programs.**

I. Introduction

Object-oriented frameworks and libraries[1] are widely used in software systems today [1]. They reduce development time and increase the user-perceived quality of programs through the reuse of existing code, such as more reliable implementations. They are used through their Application Programming Interfaces (APIs), which specify a set of functionality that client programs can use to accomplish specific tasks. As any other software artifacts, APIs are often evolved to cope with new requirements or patch security vulnerabilities (e.g., the fixing of the Heartbleed bug in OpenSSL[2]).

However, upgrading frameworks is not cost-free and may interrupt the services in worst cases. For example, the online tax filing service of Canada Revenue Agency was down for five days to patch the Heartbleed bug[3]. Raemaekers *et al.* [1] reports about the upgrade of the authentication framework of a software system that ended up consuming a whole week of work, even though developers were using automated tests to verify the upgraded system. Five days of interrupted services may cause serious losses for service providers, but five days with services having known-security-leak may expose service providers to even bigger losses. In the case of security vulnerabilities, one way to reduce the costs of framework upgrades is to perform frequent updates. In fact, framework providers usually do not patch all affected releases, but the latest one. Therefore, if the version of the framework used in client programs is too old, developers may spend a long time adapting API changes that are unrelated to the vulnerability, which in turn would slow down the upgrading process. Hence, keeping frameworks updated can reduce the reaction time to patch security leaks.

Because framework upgrades are often costly (at least in the short term), it is important for development teams to assess and forecast the cost of each framework upgrade before initiating any action. Specifically, it is important to answer the following questions: (i) what, where and how are framework APIs used? (ii) which of the used APIs are changed in the new release of the framework and how are they changed? (iii) where and how are framework APIs used in client programs? A changed API used in many locations in client programs may cause Shotgun Surgery [2], a code smell causing lots of little changes in lots of classes, which would result in high upgrade costs. Different types of API changes affect upgrade costs differently. For example, the addition of a formal parameter to an API method is easier to adapt than the removal of an API method. To adapt the latter change, developers must find a replacement of the removed method or re-implement it, while for the first case, they can simply provide a new parameter.

To assist development teams in assessing the cost of framework upgrades, we propose ACUA (API Change and Usage Auditor), a tool to collect API change and usage information through the analysis of the source or binary code of frameworks and client programs. For a given client release and a pair of framework releases, ACUA first reports which (and where) APIs are used in the client program. Second, ACUA reports the IR (infiltration ratio), *i.e,* the percentage of client program elements using framework APIs and the IIR (ideal infiltration ratio), *i.e,* the lowest infiltration ratio possible. Third, ACUA detects what (and how) APIs are changed between the currently-used and the to-be-upgraded releases of frameworks. Fourth, ACUA verifies which of the changed APIs affect the client program.

To the best of our knowledge, no previous tool has analyzed frameworks and their client programs and provided API change and usage information in such details. Yet, such a tool would help developers plan and act efficiently regarding API change adapting. For the developers of client programs, a good knowledge of how APIs are used and how they are affected by different types of API changes would help them better estimate upgrading work load and plan framework upgrades. For framework developers, such tools would help

[1]Without loss of generality, we focus on frameworks which usually provide the functions of libraries in practice.

[2]http://heartbleed.com/

[3]http://www.cra-arc.gc.ca/gncy/fq-hb-eng.html

them document the API changes that are difficult to adapt in high priority.

To assess the potential benefits of ACUA, we conduct a case study to analyze API changes and usages in 11 framework releases and their client programs, collected from a large framework ecosystem, Eclipse. With ACUA, we observed that (1) on average, more than 80% of such usage could be reduced through refactorings, (2) API change types that occur the most in frameworks and those that are used the most in the client programs are not the same, and such difference should be considered in documenting API changes and developing tools to support framework upgrading.

The remainder of the paper is organized as follows. Section II introduces background information about API change and usage. Section III describe the functionalities of ACUA and Section **??** presents the design of our case study. Section IV reports the results of the study. Section V discusses lesson learned and suggestions to developers. Section **??** discusses threats to validity while Section VI summarises the related works. Finally, Section VII concludes the paper and outlines some avenues for future works.

II. Background

This section presents some background information about APIs changes and usages.

A. API changes

Frameworks provide their services through APIs. These APIs may change during the evolution of frameworks. Des Rivières [3] has summarized API changes, based on entities changed in APIs, *e.g.,* packages, classes, modifiers *etc.*. He also pointed out which API changes may cause binary incompatibility, *i.e.*, the class files of the new releases of frameworks cannot be linked to client programs without recompilation. Although an API change may only cause binary or source incompatibility [4], most of the API changes listed by Des Rivière also cause source incompatibility, *i.e.*, there are errors when the client program source code are compiled with the new releases of frameworks. In this study, we do not distinguish between these two types of incompatibilities because, as stated by Buchholz, [5] *"Every change is an incompatible change" (a risk/benefit analysis is always required)* and hence client developers should be informed of all API changes causing source or binary incompatibility for their code.

In the classification of API changes proposed by Des Rivières, we selected those related to classes, interfaces and methods, because they are the fundamental entities of object-oriented programming languages. We summarise Des Rivières' API changes into 23 categories, among which, 15 are at reference type level (shown in Table I) and 8 are at method level (shown in Table II).

These types of API changes do not have the same effect on client programs; some of them are more difficult to adapt than the others. For example, the addition of a new `int` parameter to an API method is easier to adapt than the removal of an API method. To adapt the latter change, developers must find a replacement of the removed method or re-implement it. Knowing the types of API changes is important for accurate estimations of programs upgrade workloads.

TABLE I. Framework API change types - reference type level

ACUA	Des Rivières
Non-Existing Class (NEC)	Delete API type from API package
Non-ExistingInterface (NEI)	
Moved Class (MC)	
Moved Interface (MI)	
Decrease Access (DA)	Change public type in API package to make non-public
Change Type Kind (CTK)	Change kind of API type (class, interface, enum, or annotation type)
Expand Super Interface Set (ESIS)	Expand superinterface set (direct or inherited)
Contract Super Interface Set (CSIS)	Contract superinterface set (direct or inherited)
Add Method To Interface (AMTI)	Add API method to Interface
Add Abstract Method (AAM)	Add Abstract API method to class
Contract Super Class Set (CSCS)	Contract superclass set (direct or inherited)
Change To Abstract (CTA)	Change non-abstract to abstract
ChangeToFinal (CTF)	Change non-final to final
Change Type Bound (CTB)	Add, delete, or change type bounds of type parameter
	Add type parameter
	Delete type parameter
	Re-order type parameters
With Changed Method (WCM)	Add API method
	Delete API method
	Move API method up type hierarchy
	Move API method down type hierarchy
	All method level changes

TABLE II. Framework API change types - method level

ACUA	Des Rivieres
Non-Existing Method (NEM)	Change method name
	Delete API Method
	Move API method up type hierarchy
	Move API method down type hierarchy
Change Formal Parameter (CFP)	Add or delete formal parameter Change type of a formal parameter
Change Return Type (CRT)	Change result type (including void)
Decrease Method Access (DMA)	Decrease access: from public access to protected, default, or private access from protected access to default or private access;
Change Method To Abstract (CMTA)	Change non-abstract to abstract
Change Method To Final (CMTF)	Change non-final to final
Change Method To Static (CMTS)	Change static to non-static
Change Method To Non Static (CMTNS)	Change non-static to static

B. API Usages

To access framework APIs, client developers may extend a class, implement an interface, use framework reference types (or their subtypes as generic types, method return types or formal parameter types), or call methods defined in frameworks. These different forms of APIs usages have an impact on the process of adapting client programs to new releases of frameworks. Class extensions and interface implementations are two inheritance-style usages that require a good knowledge of the internal implementation of frameworks and API changes impact more client programs [6]. On the one hand, it is not possible to eliminate API inheritance-style usages completely. Frameworks are designed for the purpose of inversion of control (IOC) [7], *i.e.*, client programs become a part of frameworks by overriding methods in classes or interfaces provided by the frameworks. On the other hand, framework classes not designed for IOC can be replaced by composition-style usage.

Composition-style usages encapsulate framework APIs and use local APIs to hide them, while inheritance-style usages retain framework APIs. An example of composition-style usage is the use of framework reference types as private members in client classes while only accessing them within method bodies.

In general, API changes in inheritance-style usage are more difficult to adapt because they require a good knowledge of the internal implementation of frameworks. Inheritance-style usages are not avoidable in client programs for the purpose of IOC, but those not for IOC can be converted to composition-style usage.

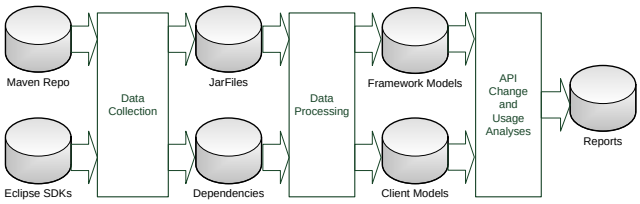


Fig. 1. ACUA Modules

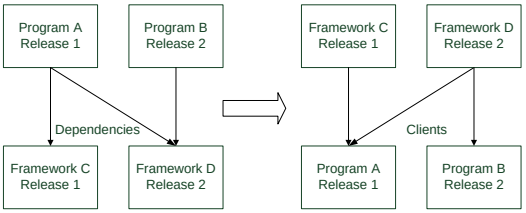


Fig. 2. Reverse dependency

III. ACUA

This section describes the modules of ACUA (shown in Figure 1) and the algorithms used to analyze frameworks API changes and API usages in client programs. The instructions to use ACUA in on our web page[4].

A. Data Collection

ACUA takes two types of inputs: Maven POM (Project Object Model) configuration files and Eclipse plugin jars.

Maven is a project management tool from the Apache Software Foundation. It uses POM files to describe the dependencies between client programs and frameworks[5]. ACUA uses the information in POM files to downloaded the jar files of the corresponding versions of client programs and the frameworks used by them, from the Maven repository.

The dependency information of Eclipse plugins are managed in two ways. From Eclipse 1.x to 3.0, the required plug-ins are specified under the "requires" node in plugin.xml contained in the plug-in folders. From Eclipse 3.1 to 4.x, the plug-in dependencies are configured in the "Require-Bundle" or "Import-Package" sections in the MANIFEST.MF files of the jar files. ACUA analyze the dependency information of Eclipse plugins, converts it into POM files and install the generated POM files and corresponding plugin jars into the Maven repository. So, original Maven projects and Eclipse plugins can be processed uniformly in the next step.

B. Data Processing

With the version information in POM files and the corresponding jar files downloaded from the Maven repository, ACUA build framework-clients maps based on the client-frameworks maps in the POM files, as shown in Figure 2.

Then, we parse the jar files of each client programs and framework release to build a meta-model. This meta-model contains reference types, method definitions, call and inheritance graphs of framework and client program releases. Our meta-model building tool is based on the ASM Java bytecode analysis framework[6].

[4]http://www.ptidej.net/downloads/replications/scam2014

[5]http://maven.apache.org/

[6]asm.ow2.org

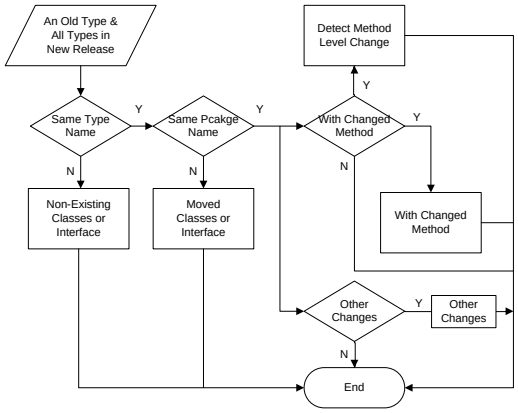


Fig. 3. Part of type-level API change detection algorithm

C. API Change Analysis

To analyze reference type level (classes, interfaces etc.) API changes, we first classify the reference types of each pair of framework releases into four categories: Stable in old release, Changed in old release, Stable in new release, Changed in new release. Here, *Stable* stands for existing in both releases and *Changed* means existing only in one release, according to their package names and reference type names, without considering type kind (class or interface), modifiers, and generic type parameters. So, Stable reference types in both releases may have differences in their methods and Changed reference types may have counterparts with the same methods.

Then, we check if the changed types in old release have counterparts in the new release with the same type names, but different package names. Those having a counterpart are classified as Moved types (classes, interfaces etc.) and the rest are classified as Non-existing types.

Next, we check for stable types between the two releases and detect the types of API changes other than Non-Existing Classes and Interfaces. The flowchart of (part of) the reference type level API change type detection algorithm is shown in Figure 3. The full algorithm is available on our Web page[4].

For the stable reference types between two releases, ACUA detects method level API changes as follows: First, ACUA checks if there is another method with the same name in the old release of the framework. If there is not, ACUA classify the method as a Non-Existing Method. If there is one, ACUA check if the method level API changes other than Non-Existing Method happened to the method. The flowchart of (part of) the method level detection algorithm is shown in Figure 4 while the full version of the algorithm is available on our Web page[4].

A changed API can be tagged with more than one change types. For example, added field to interface and added method to interface can happen to the same interface.

D. API Usage Analysis

ACUA checks which (and how) APIs are used by analyzing APIs call and inheritance graphs. It reports the following usage types: extensions of framework classes, implementations of framework interfaces, overrides of framework methods, and invocations of framework methods. For each usage type, as shown in Figure 5, ACUA collects which APIs are used, if

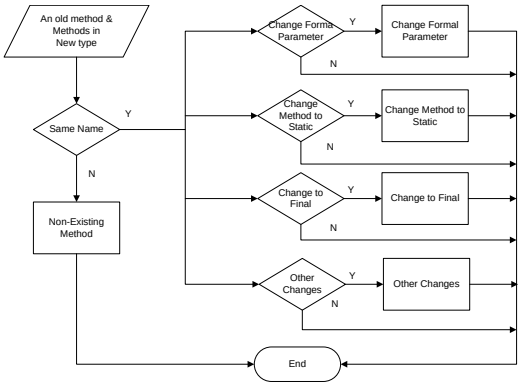


Fig. 4. Part of method-level API change detection algorithm

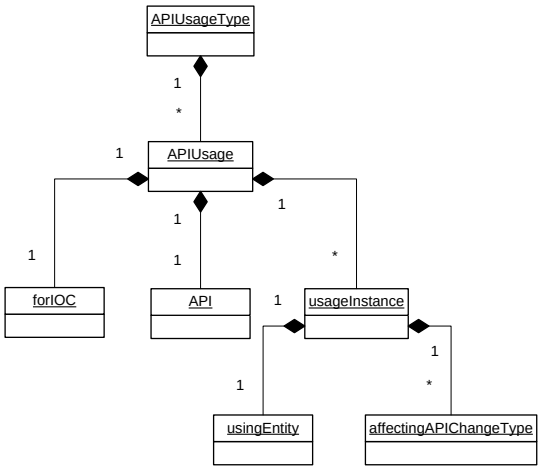


Fig. 5. API usage information

they are used for inversion of control, where they are used and the types of API changes by which they are influenced.

E. Result Outputs

After API change and usage analysis, ACUA outputs the results in XML format to facilitate reporting and further processing. The examples of API change reports and API usage reports are shown in Figure 6 and Figure 7, respectively.

IV. Case Study Results

We conducted a case study to assess the potential benefits of ACUA. Our data set contains 84 internal and 23 third-party

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Incompatibilities>
    <programName>research.org.eclipse:jdt.ui</programName>
    <oldVersion>2.1.0</oldVersion>
    <newVersion>3.0.0</newVersion>
    <incompatibilities>
        <type>AddAbstractMethod</type>
        <instances>
            <level>type</level>
            <oldAPI> class org.eclipse.jdt.internal.ui.
            javaeditor.JavaEditor</oldAPI>
        </instances>
...
    </incompatibilities>
...
</Incompatibilities>
```

Fig. 6. API change outputs

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<exposureReport>
    <clientName>research.org.eclipse:ant.ui</clientName>
    <clientVersion>3.1.0</clientVersion>
    <frameworkName>
    research.org.eclipse:core.runtime
    </frameworkName>
    <oldVersion>3.1.0</oldVersion>
    <newVersion>3.2.0.v20060603</newVersion>
    <apiUsages>
        <type>EXTENSION</type>
        <apiUsages>
            <oldAPI> class org.eclipse.core.runtime.
            PlatformObject</oldAPI>
            <invocatedInFramework>
            false
            </invocatedInFramework>
            <instances>
                <entities>
                class org.eclipse.ant.internal.ui.launch
                Configurations.AntProcess
                </entities>
            </instances>
...
        </apiUsages>
...
    </apiUsages>
...
</exposureReport>
```

Fig. 7. API usage outputs

client program releases for 11 Eclipse framework releases. Here we present and discuss the information the developers can obtain directly with ACUA.

A. Infiltration Ratio

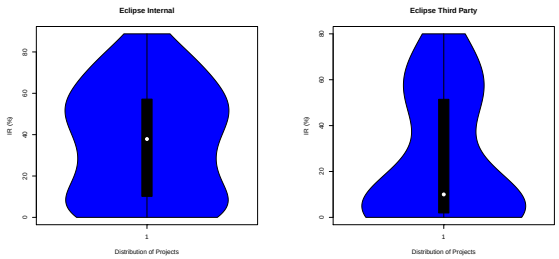
We define Infiltration Ratio (IR) as a metric to reflect the wideness of API usage in client programs. The lower IR is, the easier for developers to adapt API changes. In the equation, $RefTypes$ stands for reference types including classes and interfaces.

$$IR = \frac{\#RefTypes\ Using\ APIs}{\#Total\ RefTypes}$$

We also define Ideal Infiltration Ratio (IIR) to represent the lowest IR that a client program could reach. Here, $\#RefTypes\ For\ IOC$ is computed by the number of reference types in client programs which override framework methods called inside frameworks. The IOC reference types must use framework API through inheritance. For the other types of API usages, developers could encapsulate them with small number of local classes. Therefore, the practical lowest IR should be slightly larger than IIR, because we need to consider the encapsulating classes.

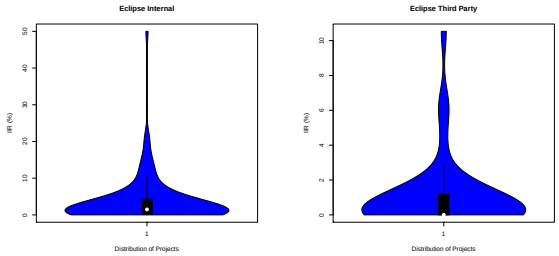
$$IIR = \frac{\#RefTypes\ For\ IOC}{\#Total\ RefTypes}$$

Figure 8 shows the IRs of internal and third-party client programs of Eclipse frameworks. We can see that internal client programs have higher IRs than third-party client programs in general, but both can reach 100% which means that all the reference types of client programs could be affected by API changes. On average, the IRs are 37% and 26% for internal and third-party client programs, respectively. If we compare IRs with IIRs shown in Figure 9, we can find that the IRs can be reduced. On average, the differences between IRs and IIRs are 33% and 25% in internal and third-party client programs, respectively. In total, the average IR and IIR of all the client programs in our study are 29% and 2%. From



(a) Eclipse Internal (b) Eclipse Third Party

Fig. 8. Infiltration Ratios



(a) Eclipse Internal (b) Eclipse Third Party

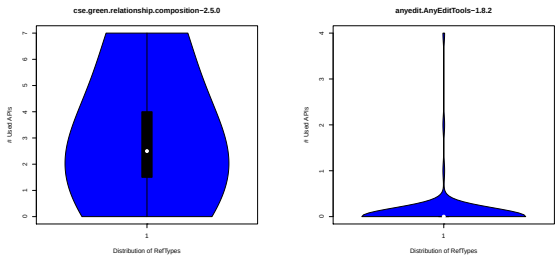
Fig. 9. Ideal Infiltration Ratios

the data, we can see more than 90% of API usages can be reduced.

For a specific client program, besides IRs, ACUA can also report the distributions of the number of APIs used by each reference types in client programs. As shown in Figure 10, the Kisses shape of the usages of APIs from eclipse.ui.editor v3.2 in anyedit.AnyEditTools-1.8.2 is preferred over the vase shape of API usages form eclipse.jdt.core v3.2 in cse.green.relationship.composition v2.5.0. The former has less reference types using large number of APIs than the latter.

B. API change distribution in frameworks and client programs

Figure 11 and 12 show the distributions of reference type and method level API change types in Eclipses frameworks. At reference type level, the top four API change types (WithChangedMethod, NonExistingClass, ContractSuperClass, and ExpandSupterInferfaceSet) represent about 82% of total changes . At method level, the top four API change types (NonExistingMethod, ChangeFormalParameter, ChangeReturnType, DecreaseMethodAccess) cover 98% of total method level API changes.



(a) cse.green.relationship.composition-2.5.0 (b) anyedit.AnyEditTools-1.8.2

Fig. 10. Project API Infiltration

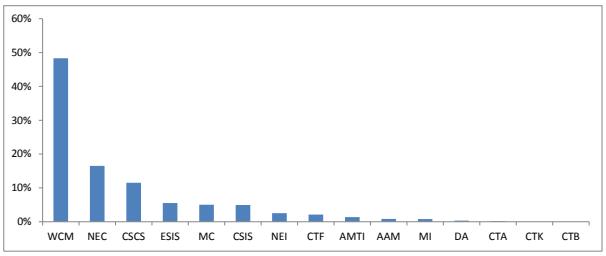


Fig. 11. Reference type level API changes

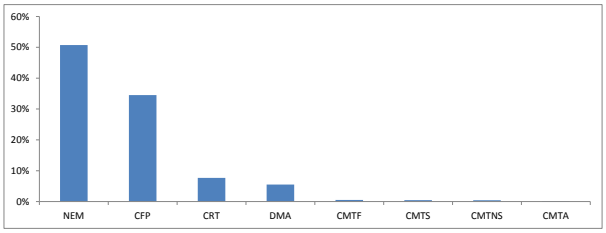


Fig. 12. Method level API changes

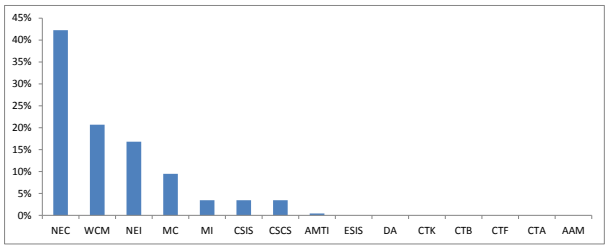
We find that the API changes used more often in client programs are not the same as the API changes in frameworks. Figure 13 shows that the top-2 API change types are the same as top-2 API change types in frameworks (WithChangedMethod and NonExistingClass) with cases of switching 1st and 2nd positions. ContractSuperClasses and ExpandSuperInterfaceSet are the 3rd and 4th API change types with 12% and 6% of P_T values. In third party client programs, MovedInterface, ContractSuperInterfaceSet take the 3rd and 4th positions with 11% and 9% P_{T_U}. ContractSuperClasses falls to 5th with 6% P_{T_U}. In internal client programs, NonExistingInterface and MovedClass are the 3rd and 4th used API changes with 17% and 9% P_{T_U}. In frameworks, ContractSuperInterfaceSet and MovedClass are 5% of total API changes at reference type level, while NonExistingInterface is 3% and MovedInterface is 1%. Same as Apache, ExpandSuperInterfaceSet does not affect client programs either.

At method level, the top-3 API changes used in client programs are the same as those in frameworks for both ecosystems: NonExistingMethod, ChangeFormalParameter, and ChangeReturnType, as shown in Figure 14.

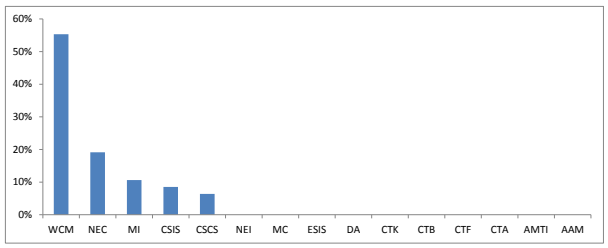
In general, with ACUA, we can see that NonExistingClass and NonExistingMethod are in the top-2 changes and they require additional search effort to find the replacements during the upgrading process. Some reference type level API change types, such as NonExistingInterface, they are more often used in client programs than their proportions in total API changes. Framework developers should avoid such changes during evolution or provide detailed documents when they are necessary. Client program developers should reduce IRs to alleviate the impact of API changes.

V. Discussion

Compilers are basic tools that report API change impacts in client programs and developers still need them while conducting concrete upgrading tasks to verify changes and generate binary code. However, compilers cannot help developers

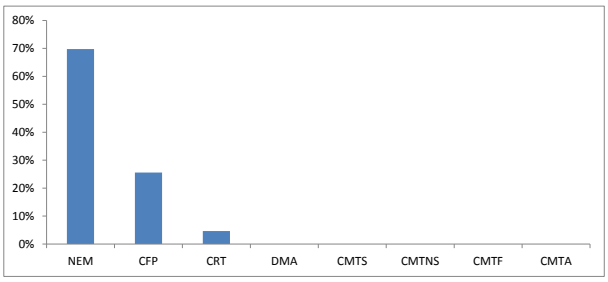


(a) Eclipse Internal

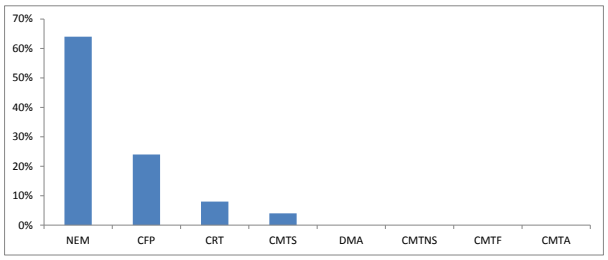


(b) Eclipse Third Party

Fig. 13. Reference type level API changes of Eclipse frameworks used in client programs



(a) Eclipse Internal



(b) Eclipse Third Party

Fig. 14. Method level API changes of Eclipse frameworks used in client programs

much to plan and estimate upgrading workloads. Traditional compilers do not provide the following information that ACUA reports:

a) API change type:: compilers can detect API changes by showing compiling errors, but they do not report which types of API changes caused the errors. Because API changes are not equally difficult to adapt, such information is useful to plan program upgrade effectively. ACUA analyze client programs and two releases of frameworks to generate API usage report that summarize which APIs are used , where they are used (in client programs) and the types of API changes affecting them. With API usage reports, developers have a clearer picture of API change impacts.

b) API infiltration:: compilers cannot report how widely API are used in client programs. APIs keep evolving, developers should know how seriously APIs infiltrate in their client programs to prepare for future API changes. If large numbers of the APIs of a framework widely used in client programs, it will be difficult to adapt to major changes in the frameworks or to switch to other frameworks with similar functions. ACUA can compute IR and IIR to tell developers the current API infiltration ratio and its possible lower boundary. The API usage data collected by ACUA can also be used to visualize API infiltration, as shown in Figure 8.

VI. Related Work

Many existing tools help developers on framework API change and usages from different perspectives. CatchUP! [8] records the refactoring operations in one release and replay them in another. AURA [9], Diff-CatchUp [10], HiMa [11], and SemDiff [12] generate the API change rules between two releases of frameworks. Twinning [13] adapts different Java frameworks with similar functionalities. MAN [14] maps APIs between Java and C#. Portfolio [15] searches and visualizes relevant functions and their usages from an internal database. LibSync [16] helps developers learn complex API usage change patterns from the clients that have been already upgraded to new releases of frameworks. Change Distilling [17] rebuilds change road-maps between two releases of programs. LSdiff [18] summarizes the changes between two releases of frameworks into systematic structural differences and presents anomalies in them. Ref-Finder [19] detects the 63 types of refactoring. MADMatch [20] matches evolving program elements with an approach that uses Error Tolerant Graph Matching algorithm. Exapus [21] explores API usages from different views. In this paper, we present ACUA to report both API changes and usages in frameworks and client programs in details.

VII. Conclusion

When frameworks evolve, APIs may change between releases. Nowadays, patching security vulnerabilities of programs becomes an important reason to upgrade frameworks and should be done as soon as possible. Usually, framework providers only fix security leaks in the latest releases. If the releases of frameworks used are much behind , applying security patches may be delayed by adapting non-security-related API changes. Therefore, keeping framework updated should be considered as a regular task in software maintenance. In this paper, we propose a tool ACUA to generate API change and usage reports by analyzing the source or binary code of both frameworks and clients programs in Java. These reports help developers plan proactive framework upgrading. To assess the benefits of ACUA, we conduct a case study with 11 framework releases and their client program releases from a framework ecosystem, Eclipse. With the reports generated by ACUA, developers can know how API are used in and which API changes affect their client programs. Such information can help them to estimate upgrading work load and plan upgrading tasks. Empirical studies to quantitatively evaluate the benefits of ACUA and tools using the reports generated by ACUA to directly help framework upgrading are future works.

ACKNOWLEDGMENTS

We thank Daniel German for providing the Maven central repository snapshots. This work has been partly funded by the NSERC Research Chairs on Software Patterns and Patterns of Software and on Software Change and Evolution.

REFERENCES

[1] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ser. ICSM '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 378–387.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] D. Rivières, "Evolving java-based apis 2," 2008. [Online]. Available: http://wiki.eclipse.org/Evolving_Java-based_APIs_2

[4] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in java programs caused by library upgrades," in *CSMR-WCRE*, 2014, pp. 64–73.

[5] M. Buchholz, "Kinds of compatibility: Source, binary, and behavioral," 2008. [Online]. Available: https://blogs.oracle.com/darcy/entry/kinds_of_compatibility

[6] J. Bloch, *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.

[7] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support api evolution," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 274–283.

[9] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848

[10] Z. Xing and E. Stroulia, "API-evolution support with diff-CatchUp," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818 – 836, December 2007.

[11] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *Proceedings of 34th International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 353–363.

[12] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011.

[13] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 205–214.

[14] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 195–204.

[15] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 111–120.

[16] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321.

[17] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 725–743, November 2007.

[18] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.

[19] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 371–372. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882353

[20] S. Kpodjedo, F. Ricca, P. Galinier, G. Antoniol, and Y.-G. Guéhéneuc, "Madmatch: Many-to-many approximate diagram matching for design comparison," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1090–1111, 2013.

[21] C. D. Roover, R. Lammel, and E. Pek, "Multi-dimensional exploration of api usage," in *ICPC*, 2013, pp. 152–161.