Empirical Software Engineering manuscript No.
(will be inserted by the editor)

An Exploratory Study of API Changes and Usages based on Apache and Eclipse Ecosystems

**Wei Wu · Foutse Khomh · Bram Adams ·
Yann-Gaël Guéhéneuc · Giuliano Antoniol**

Received: date / Accepted: date

Abstract Frameworks are widely used in modern software development to reduce development costs. They are accessed through their Application Programming Interfaces (APIs), which specify the contracts with client programs. When frameworks evolve, API backward-compatibility cannot always be guaranteed and client programs must upgrade to use the new releases. Because framework upgrades are not cost-free, observing API changes and usages together at fine-grained levels is necessary to help developers understand, assess, and forecast the cost of each framework upgrade. Whereas previous work studied API changes in frameworks and API usages in client programs separately, we analyse and classify API changes and usages together in 22 framework releases from the Apache and Eclipse ecosystems and their client programs. We find that (1) missing classes and methods happen more often in frameworks and affect client programs more often than the other API change types do, (2) missing interfaces occur rarely in frameworks but affect client programs often, (3) framework APIs are used on average in 35% of client classes and interfaces, (4) most of such usages could be encapsulated locally and reduced in number, and (5) about 11% of APIs usages could cause ripple effects in client programs when these APIs change. Based on these findings, we provide suggestions for developers and researchers to reduce the impact of API evolution.

1 Introduction

Object-oriented frameworks 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, which should be reliable and stable [2]. As an added benefit, frameworks are updated by their developers [3] to cope with new requirements or patch security vulnerabilities, while the users of frameworks can focus on their own client programs.

Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, Giuliano Antoniol,
DGIGL, École Polytechnique de Montréal, Canada

However, client-program developers must also upgrade regularly to new releases of frameworks in addition to maintaining their own programs. New releases of frameworks include bug fixes and new features and also protect client programs from security issues. These framework upgrades can be costly: Raemaekers *et al.* [1] report 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 their upgraded client program. In Linux Debian, it took developers seven weeks to upgrade Perl from 5.10 to 5.12[1].

Frameworks are used by client programs through their Application Programming Interfaces (APIs), which specify sets of functionalities. Adapting to API changes is the first task that client-program developers must perform in the upgrading process. This task—and its cost—depends on both the types of API change, *i.e.,* which and how APIs are changed between two releases of a framework, and on the API usages in client programs, *i.e.,* how and how much these changed APIs are used.

For example, developers can easily adapt their client programs to a new framework version that moved a class from one package to another, if it is used only locally in one method, where the opposite holds for a new version that removes of a class that is being used as parameter type in dozens of methods. To adapt to the latter change, developers must find a replacement for the removed class, re-implement it, or modify all their methods, while to adapt to the former, they only need to update the package name of the class in one file. This example shows that, given the sheer numbers and sizes of today's frameworks and client programs and given the sizes and distributions (space, time) of today's development teams, developers face challenges assessing the impact of API changes.

Hence, developers need tools to collect facts about both API changes between framework releases and API usages in their client programs to assess API changes and upgrade costs. However, to the best of our knowledge, most previous works and existing tools focus either on API changes or API usages, separately. Des Rivières[2] discussed in detail API contract compatibility and classified API changes according to the elements of Java APIs, such as package, class, method, and so on. However, he did not investigate the distributions of API change types in client programs. Dietrich *et al.* [4] investigated the differences between Java compile-time and link-time compatibility in the Qualitas corpus [5] and reported that such incompatibilities exist but affect client programs rarely. Businge *et al.* and others [6–8] studied API usages but did not consider potential changes in these usages caused by framework evolution. Existing tools help developers either for API change detection, (*e.g.,* Java API Compliance Checker[3]) or API usage detection (*e.g.,* Tattletale[4]), but not both. Only few works studied API changes and usages together. Besides Dietrich *et al.* [4], Robbes *et al.* [9] conducted a study on how developers react to API deprecation in the Smalltalk Squeak/Pharo ecosystem. Their work is limited to one specific API change.

Therefore, we propose ACUA (API Change and Usage Auditor) [10] to analyse API changes and usages together in Java frameworks and client programs. On the

[1] `https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=619117`
[2] `http://wiki.eclipse.org/Evolving_Java-based_APIs_2/`
[3] `http://ispras.linuxbase.org/index.php/Java\_API\_Compliance\_Checker`
[4] `http://tattletale.jboss.org/`

one hand, ACUA uses Des Rivières' definitions of API change types[2], considering public and protected Java software entities, such as classes, interfaces and methods, as APIs.

Since the three types of APIs are the most important program entities [11], ACUA detects their change types instead of the whole Des Rivières' classifications. On the other hand, ACUA considers five possible usages of framework APIs inside the APIs of a client program: (1) extending framework classes, (2) implementing framework interfaces, *and* using framework reference types either as (3) generic types in templates, (4) method return types, and–or (5) formal parameter types.

We call these five types of API usages *API-injection usages* because framework APIs become *part* of ("are injected into") the APIs of the client programs. API-injection usages are different from *local API usages*, *e.g.,* in method bodies as illustrated in the example before. Upon changes to the framework APIs, local API changes would require only local changes to the client-program method bodies while API-injection usages would require changes to any client methods *using* the methods and–or reference types of the client programs that depend on the framework APIs. In other words, these changes would propagate through the API-injection usages to other parts of the client programs and hence potentially would be more costly.

ACUA can report which APIs are changed between two releases of a framework are used in client programs, where and how this is done. With these lists of used APIs and categorised API changes, developers can assess the extent of the changes needed to adapt their client programs, while researchers can know which types of API change types impact more frequently client programs and, hence, propose mitigating techniques: we will discuss how encapsulating framework APIs with composition [12] could reduce the impact of API changes and how existing API change-rule building approaches, such as SemDiff [13] and AURA [14], can help developers find replacements to missing classes and methods.

We apply ACUA on 22 framework releases within two ecosystems, Apache and Eclipse, and their internal and third-party client programs. We find that the most common types of API change are missing classes and methods. We also find that those two changes affect client programs more often than the other types of API changes. The other common API change types in frameworks differ from those affecting client programs the most, which is important when documenting API changes or developing tools to support framework upgrading.

On average, in our data set, the APIs of a framework are used by 35% of its client classes and interfaces, but most of such usages could be eliminated by applying certain design patterns, such as Adapter and Facade [2]. About 14% and 8% of the APIs in the Apache and Eclipse frameworks in our data set, respectively, are injected in the client program APIs. Also, we find that there is a correlation between API changes and API-injection usages in Apache internal client programs where Add Abstract Method, Moved Interface, and Change Type Kind are the three API change types that impact framework APIs injected into client program APIs.

The remainder of the paper is organised as follows. Section 2 introduces background information. Section 3 describes ACUA, the tool used in the exploratory study. Section 4 presents the data sets and reports the results of the study. Section 5 discusses threats to validity while Section 6 summarises and contrasts our work with previous work. Finally, Section 7 concludes and introduces future work.

2 Background

In this section, we first present the background information about API changes and usages, the definitions, and the statistical tests related to this study and introduce the research questions in the end.

2.1 API Changes

Frameworks provide their services through APIs. These APIs may change during the evolution of frameworks. Based on the Java Language Specification[5] and his experience within the development of Eclipse, Des Rivières[2] reported whether an API change may cause binary incompatibility (*i.e.,* the class files of a client program cannot be linked to the new release of a framework without recompilation) or source incompatibility (*i.e.,* there are errors when the client program source code is compiled against the new release of a framework).

Most API changes cause both binary and source incompatibilities. Only in a few particular cases, either binary-only or source-only incompatibilities appear, as discussed by Dietrich *et al.* [4]. For example, if an API method in a framework changes its return type in the new release to a subtype of the current return type, the change would be source compatible but binary incompatible, because the Java language specification (like those of other programming languages: C++, D, Scala...) allows the covariance of return types but the JVM uses the exact return type during the linking process. An example of binary compatible but source incompatible API changes is adding a checked exception to an API method, because the JVM does not consider exceptions as a part of the method descriptors during the linking process, while the Java compiler checks them during the compilation process.

A preliminary analysis of the changes in the two ecosystems of interest in this paper, Apache and Eclipse, convinced us that such binary-incompatible-only changes are rare (less than 5% of total changes), hence we do not consider distinguishing these two types of API changes.

Des Rivières[2] categorised API changes into 149 types. Previous approaches and tools chose a subset of these API changes, such as the one by Dig *et al.* [15] (seven changes types, including class and method renames) or Java ACC[6] (16 changes types, including field and method modifier changes). Among the 149 change types, we choose to focus on changes pertaining to classes, interfaces, and methods because these three types of entities are fundamental to object-oriented programming [11] and their incompatible changes from one release to the next would break client programs. We reorganised these API changes into 23 categories according to their potential impact on the developers of client programs. For example, we split *Delete API type from API package* into *Missing API Type* and *Moved API Type* because any modern IDE can help developers locate the replacements of the latter but none can help systematically with the former.

Among the 23 considered API change types, 15 are at the class/interface level, shown in Table 1, while the remaining eight are at the method level, shown in Table

[5] `https://docs.oracle.com/javase/specs/jls/se7/html/jls-13.html`
[6] `http://ispras.linuxbase.org/index.php/Java\_API\_Compliance\_Checker`

Table 1: Description of framework API change types - class/interface level

ACUA	Des Rivières
MSC (Missing Class)	Delete API type from API package
MSI (Missing Interface)	
MVC (Moved Class)	
MVI (Moved Interface)	
CSIS	Contract SuperInterface Set (Remove interfaces from the current type or parent types)
ESIS	Expand SuperInterface Set (direct or inherited) (Add interfaces to the current type or parent types)
CTB (Change Type Bound)	Add, delete, or change type bounds of generic type parameter
	Add generic type parameter
	Delete generic type parameter
	Re-order generic type parameters
CMC (Containing Method-level Changes)	Containing method level changes in Table 2
CTK	Change Type Kind of APIs: (class, interface, enum, or annotation type)
DA	Decrease Access: make public type in API package non-public
AAM	Add Abstract API Method to class
CTF	Change non-final To Final
CTA	Change non-abstract To Abstract
CSCS	Contract SuperClass Set (Remove super class from the current type or parent types)
AMTI	Add API Method to Interface

Table 2: Description of framework API change types - method-level

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

2, as part of the CMC (Containing Method-level Changes) category at class/interface level. We do not detect method-level API changes accompanying the change types MSC (Missing Classes) and MSI (Missing Interface), because there is no reliable way to automatically locate the counterparts of these classes and interfaces in the new releases of frameworks.

2.2 Measuring the Impact of API Changes

The above types of API changes do not have the same impact on the developers of client programs. For example, adapting to the addition of a new `int` parameter to an API method is different from adapting to the removal of an API method, because for the latter, developers must find a replacement for the removed method or reimplement it. Therefore, we define $P_{<T>}$, as the percentage of API changes in

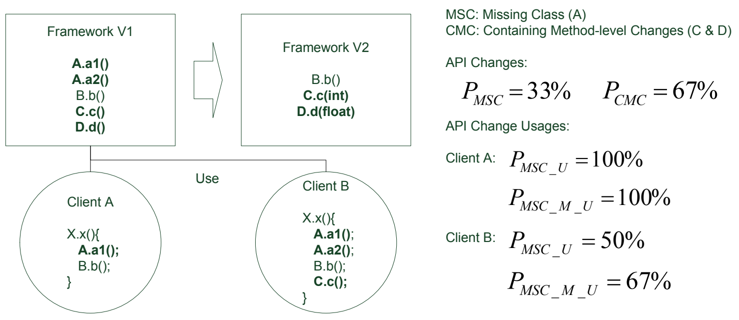


Fig. 1: Measurement computation

a new framework version that are of type T, where T is a place holder for the API change types defined in Tables 1 and 2. To calculate $P_{<T>}$, we count the number of changed APIs $\#T$ for each type T and the total number of changed APIs across all types $\#ALL$. Finally, we compute $P_{<T>}$ as follows:

$$P_{<T>} = \frac{\#T}{\#ALL} \tag{1}$$

Figure 1 shows an example of this calculation for one MSC (Missing Class) and two CMC (Containing Method-level Changes). Therefore, the percentage of API change types for MSC and CMC are $P_{MSC} = 33\%$ and $P_{CMC} = 67\%$, respectively.

To observe the effects of each type of API change T on a client program, we use the metric $P_{<T>\_U}$, which is similar to $P_{<T>}$ but only considers the changes to APIs affecting a client program. This yields a measure of the impact of each API change type on client programs, computed using Equation (2).

$$P_{<T>\_U} = \frac{\#T\_U}{\#ALL\_U} \tag{2}$$

where $\#T\_U$ is the number of changed APIs of Type T used by client programs and $\#ALL\_U$ is the number of all changed APIs used by client programs.

Figure 1 also shows an example of $P_{<T>\_U}$ calculations. In this example, client `A` uses only one changed API affected by MSC, while client `B` has two and is affected by CMC additionally. At class level, MSC affects both clients A and B once. Therefore, P_{MSC_U} is 100% for `A` and P_{MSC_U} is 50% for `B`. However at method-level, the effect on B is more serious than that on A, because B uses two methods while A only uses one.

To measure the effect of class/interface-level API changes of the type T on client programs at method level, we define the metric $P_{<T>\_M\_U}$, which is similar to $P_{<T>\_U}$, , but counts each individual method usage of a changed class/interface-level API. Hence, we compute $P_{<T>\_M\_U}$ using Equation (3).

$$P_{<T>\_M\_U} = \frac{\#T\_M\_U}{\#ALL\_M\_U} \tag{3}$$

In the example in Figure 1, $P_{MSC\_M\_U}$ becomes 67% for client B while $P_{MSC\_M\_U}$ is still 100% for client A.

These metrics will help understand the relative frequency of different kinds of API changes and their impact on the API clients.

```
// Framework
class FrameworkClass {
        public void method(FrameworkInterface frameworkInterface)
        { frameworkInterface.methodToImplement(); } }

interface FrameworkInterface{
        public void methodToImplement(); }

// Client - API usage types

class U1 implements FrameworkInterface{
        public void methodToImplement(){...}  }

class U2 extends FrameworkClass{...}

class U3 <T extends FrameworkClass> { T u3(){...} }

class U4 { FrameworkClass u4(){...} }

class U5 {
        void u5(FrameworkClass frameworkClass){
                // U6 triggered by U5
                frameworkClass.method(...); } }

class U6 {
        private FrameworkClass frameworkClass = new FrameworkClass();
        void method_wrapper(){
                frameworkClass.method(...);} }

// Client - API usage type effects

class OtherUsage {
        // Directly affected by the changes to FrameworkClass.method
        void method_affected(U2 u2){
                u2.method(...);}

        // Protected from the changes to FrameworkClass.method
        void method_protected(U6 u6){
                u6.method_wrapper(...);} }
```

Fig. 2: API usage example

Table 3: Description of framework API usage types

Type	Framework API Usage	API-Injection	Encapsulable
U1	Inheritance for inversion of control	Yes	No
U2	Optional inheritance	Yes	Yes
U3	As generic types	Yes	Yes
U4	As method return types	Yes	Yes
U5	As method formal parameter types	Yes	Yes
U6	In method implementations	No	Yes

2.3 API Usages

Developers use framework APIs in different ways. For example, frameworks can be designed for the purpose of inversion of control (IOC) [2], *i.e.,* client programs must override or implement methods in the classes or interfaces defined in the frameworks. Alternatively, some frameworks can also be used as libraries, *i.e.,* client programs can directly instantiate framework classes and invoke their methods.

Certain API usages expose client programs to API changes more than the others. Class extensions and interface implementations are two inheritance-style usages that require the understanding of the internal implementation of frameworks and that propagate API changes from the framework to the client programs [12]. Developers cannot avoid such usages if they want to benefit from the functionalities of the frameworks, although they could favour composition [12] over inheritance to encapsulate these usages and, thus, limit change propagation.

In Table 3, we define the six types of API usages that we want to study in relation to API changes. We do not directly consider other usages, such as using framework types as exceptions, because these usages are similar to $U3 - U5$. Each API usage type has an example with the same name in Figure 2. Types $U1$ and $U2$ are inheritance-style usages. As shown by the example in Figure 2, `U2.method()` is actually a framework API propagated around by the client program class $U2$. If those APIs change, the affected subclasses and interface implementations will propagate the changes to where they are used in the client programs, as shown in method `OtherUsage.method_affected()`.

Developers cannot eliminate API inheritance-style usages completely because of IOC. However, if a public or protected subclass, such as `U2`, extending a framework class or implementing framework interfaces, does not override any methods invoked inside the framework, it probably is an *optional-inheritance*, which can be replaced with a *composition* [12]. Such a replacement encapsulates framework APIs with client program classes to localise the changes of API and reduce their impact on the other parts of client programs. An example of composition-style usage is the use of framework classes/interfaces as private fields in client classes, only accessing their APIs within the method implementations. Class `U6` in Figure 2 illustrates a composition-style usage: `OtherUsage.method_protected()` is not affected by framework API changes because the class `U6` wraps the framework API with a client program API.

Besides inheritance-style API usages ($U1$ and $U2$), other usages ($U3 - U5$) in client program APIs also propagate framework API changes inside the client programs by using framework APIs as generic types, method return types, or formal parameter types. In $U1 - U5$ usages, these framework APIs become part of client program APIs (classes, interfaces, and methods). Thus, we define $U1-U5$ as *API-injection usages*. Among these API-injection usages, $U1$ cannot be encapsulated with composition because the client class is required to have the same API for the purpose of IOC, while $U2-U5$ could be. Therefore, we call $U2-U5$ as *encapsulable API-injection usages*. Replacing $U2 - U5$ API usages in client program APIs can help reduce API change propagation more effectively than replacing $U6$ while upgrading frameworks, because $U6$ only exists in the implementation code of client programs and does not propagate API changes.

It is worth noting that API-injection usages do not always propagate framework API changes. For example, if the methods defined in a framework class are only used inside a subclass of a client program, the usage ($U1$) is an API-injection usage, but it does not (yet) propagate the framework class changes. However, client program developers (especially new developers) may use the methods defined in the framework classes in the future because it technically is not forbidden to use them. In that case, *this usage* would propagate framework API changes. If client programs keep evolving, more and more cases of propagating API-injection usages will make framework upgrading difficult eventually.

2.4 Measuring the Impact of Usage Types

Based on the above types of usage, we define a USage Ratio (USR) metric that reflects the percentage of client program reference types which use framework APIs directly. The lower the value of USR, the easier it will be for developers to adapt API changes, since the client application is coupled less to the APIs. In the example in Figure 2, the USR of the framework in the client is 86% (six in seven classes).

$$USR = \frac{\#\mathit{Client\ Reference\ Types\ Using\ Framework\ APIs}}{\#\mathit{Total\ Client\ Reference\ Types}} \tag{4}$$

To investigate the API-injection usages in client programs, we define two metrics. First, we define the Ideal USage Ratio (IUSR) metric to compute the lowest USR value that a client program can reach while still being able to integrate into frameworks. Here, $\#Client\ Reference\ Types\ With\ U1$ is computed using the number of client reference types that override framework methods called inside frameworks, *i.e.,* API usages for IOC that cannot be encapsulated using composition. Therefore, in practice, the lowest USR value usually is slightly larger than IUSR, because there may be classes implementing the compositions for $U2-U6$ usages besides the classes of $U1$ usage.

$$IUSR = \frac{\#\mathit{Client\ Reference\ Types\ With\ U1}}{\#\mathit{Total\ Client\ Reference\ Types}} \tag{5}$$

In the example in Figure 2, `FrameworkClass` is a framework class and `FrameworkInterface` is a framework interface for IOC. The client program has seven classes. Among them, the only `U1` case is the class connecting with the framework through IOC. Therefore, the IUSR value of the framework in this client program is 14% (1/7).

Second, as shown in Table 3, $U2-U5$ inject framework APIs into client program APIs, but in contrast to $U1$, can be encapsulated by composition to localise API changes. Some cases of $U6$ may also be caused by $U2-U5$, for example, in the method `U5.u5(...)` in Figure 2. Therefore, reducing the cases of encapsulable API-injection usage is an effective way to decrease USR.

For this reason, we define the Encapsulable API-Injection Ratio (EAR) to measure the number of API usages that propagate API changes but could be avoided. A lower value of EAR means less API change propagations.

$$EAR = \frac{\#\mathit{Client\ Reference\ Types\ With\ U2-U5}}{\#\mathit{Total\ Client\ Reference\ Types}} \tag{6}$$

With these three metrics, developers can assess the impact of framework API changes. A large difference between USR and IUSR or a high EAR value would indicate that large percentages of API usages are not due to IOC and can be reduced through composition.

2.5 Measuring the Correlation between API change and Encapsulable API-injection Usage

If API changes are correlated with the encapsulable API-injection usages, it indicates that API changes are propagated in client programs and developers can control the propagating by encapsulate API usages through composition. To investigate if there is such a correlation, we first divide API usages into two groups: $U2-U5$ and $\{U1, U6\}$. Then, we check in each group what percentage of usage are affected by the different types of API changes defined in in Table 1 and Table 2. Next, we apply Pearson's Chi-squared [16] and Fisher's exact test [17] to verify if $U2-U5$ and API change are dependent. We apply both Pearson's Chi-squared and Fisher's exact tests, because the former requires a larger numbers of observations than the latter. We want to verify if there is any difference between the two tests on our data set.

If API changes and encapsulable API-injection usages are dependent, we further compute the Odds Ratios [18] of the different types of API changes to assess which changes are more likely to be applied to $U2 - U5$ than to others. An odds ratio measures the odds of an event causing an outcome in comparison to the odds of another event to cause this outcome. If the value of the odds ratio is greater than one, the one event is more likely to cause the outcome than the other event. We compute the odds of a specific API-change type applying to the encapsulable API-injection usages in comparison to its odds happening to the other change types. Depending on which change types tend to occur most with $U2 - U5$, it might be more or less interesting to refactor $U2 - U5$.

2.6 Summary

Using the concepts introduced above, we conduct a study to answer five research questions:

– RQ1: How often does each type of API change happen in frameworks? ($P_{<T>}$)
– RQ2: How often does each type of API change affect client programs? ($P_{<T>\_U}$ and $P_{<T>\_M\_U}$)
– RQ3: How widely are APIs used in the client programs? (USR)
– RQ4: What percentage of framework API usages are the API-injection usages? ($IUSR$ and EAR)
– RQ5: Which type of API change affects the encapsulable API-injection usages more often?

These research questions are related to different aspects of API evolution as shown in Figure 3. RQ1 is about API changes during framework API evolution. RQ2 is about the effect of the API changes on client programs. RQ3 is about framework API usages in client programs. RQ4 analyses the *potential* for reducing API change impact on client programs. RQ5 combines the findings of RQ4 with the distribution of change types of RQ2 to understand the *actual* degree of propagation of API changes in client programs.

Before discussing how we answer these research questions in the context of a case study on frameworks in the Apache and Eclipse ecosystems, we first present the ACUA tool [10] that we have developed to measure the metrics above.

3 ACUA

The functional modules of ACUA are shown in Figure 4. Elements with gray background are the modules of ACUA and those with white background are the

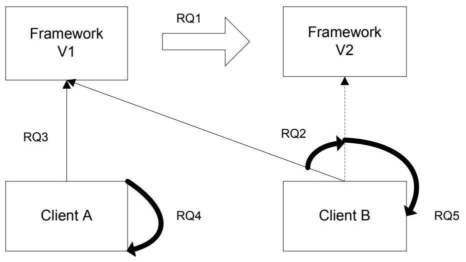


Fig. 3: Relations between research questions and API Evolution

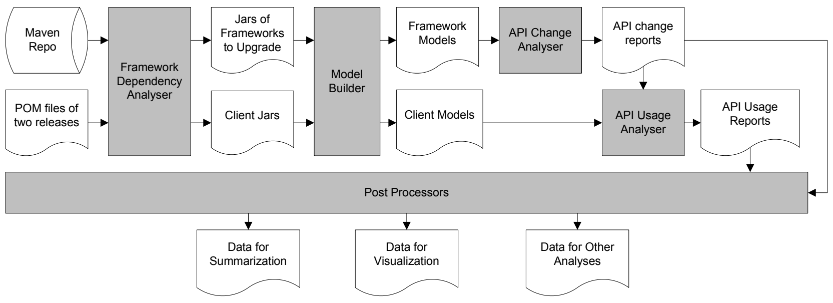


Fig. 4: ACUA Modules

inputs and outputs of each module. We detail the API change and usage detection algorithms of ACUA below. We describe the ACUA work-flow and its inputs and outputs with a running example.

3.1 Inputs

3.1.1 Maven projects

For frameworks and their client programs managed as Maven projects, ACUA takes the Maven POM (Project Object Model) configuration files of two releases of a framework as inputs, because ACUA analyses API changes between the two releases. Maven is a project management tool from the Apache Software Foundation[7]. Maven projects store the information about the dependencies between client programs and frameworks in POM files. The advantage of using Maven POM files is that Maven is widely used in software development and provides the complete information required by ACUA, such as dependencies and Jar file locations.

3.1.2 Other projects

We convert non-Maven projects with dependency information, such as Eclipse plug-ins, to Maven projects to be analysed by ACUA. For example, the dependency information of Eclipse plugins are managed in two ways. From Eclipse 1.x to

[7] `http://maven.apache.org/`

3.0, the required plug-ins are specified under the `requires` node in the `plugin.xml` file 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 of the `MANIFEST.MF` files of the Jar files. We use the two formats of dependency information in Eclipse plug-ins, convert them into POM files and install the generated POM files and corresponding plug-in Jars into the Maven repository, so that Eclipse plug-ins can be processed uniformly as Maven projects. For non-Maven programs whose dependencies do not always have version information, like projects managed by Apache Ant, ACUA users would need to complete the version information.

3.2 Framework Dependency Analyser

The Framework Dependency Analyser (FDA) uses the information in POM files to detect the changes in framework versions. FDA then connects to a remote or local Maven repository to download the Jar files of the corresponding versions of the frameworks and the client program. FDA interacts with a Maven repository through the Aether library[8].

3.3 Model Builder

Next, the Model Builder parses the Jar files of the frameworks and the client programs to build the models containing reference types, method definitions, call and inheritance relations of framework and client program releases, according to a custom meta-model. The Model builder uses the ASM Java bytecode analysis framework[9] to extract the model data from the Jar files.

3.4 API Change Analyser

Taking the models of frameworks built by the Model Builder as input, the API Change Analyser (ACA) detects the API changes between the two framework releases and classifies them into the types presented in Section 2. ACA outputs the classified API changes in the form of API change reports for the two releases of each framework.

ACA detects API changes between two releases of a framework at type (classes and interfaces) level and method-level. To analyse type-level API changes, ACA first classifies the types of two releases of a framework into four categories as shown in Table 4 and detects API change types based on these categories. The names of classes and interfaces contain their package names and type names, but ignore the kind of type (class or interface), modifiers, and generic type parameters. Hence, an interface in $O \exists B$ may become a class in $N \exists B$ with the same name. Also, a class with the same name in $O \exists B$ and $N \exists B$ may contain method-level changes.

Then, ACA checks if the classes and interfaces in $O \exists O$ have counterparts in the $N \exists N$ with the same type names, but different package names. Those having

[8] `http://www.eclipse.org/aether/`

[9] `http://asm.ow2.org/`

Table 4: Categories of classes and interfaces

Releases	Classes and interfaces	
	With same names in both releases	With names only in one release
Old	$O\exists B$ (Old version existing in both)	$O\exists O$ (Existing only in old)
New	$N\exists B$ (New version existing in both)	$N\exists N$ (Existing only in new)

such counterparts are classified as Moved types (classes, interfaces) and the rest are classified as missing types.

Next, ACA checks the classes and interfaces in $O\exists B$ and $N\exists B$ to detect the types of the other type-level API changes. Figure 5 is the flowchart of the type-level API change detection algorithm.

For classes and interfaces in $O\exists B$, ACA detects method-level API changes as follows. First, ACA checks if there is another method with the same name in $N\exists B$. If there is not, ACA classifies the method as a Missing Method. If there is one, ACA checks if the other method-level API changes happened to the method. The flowchart of the method-level detection algorithm is shown in Figure 6.

As shown in the two flowcharts, a changed API can belong to more than one API change type. For example, changes “added field to interface” and “added method to interface” can happen to the same interface.

3.5 API Usage Analyser

Based on the models of client and the API change reports generated by ACA, the API Usage Analyser (AUA) detects which, where, and how the APIs of the frameworks are used in the client programs, then verifies if the used APIs are changed in the new releases of the frameworks and/or affected by some types of API changes. As the output, AUA generates API usage reports for each framework including how the used APIs are affected by API changes.

AUA checks which and how APIs are used by analysing API type inheritance and API method invocations in client programs. It reports the usage types $U1-U6$ mentioned in Table 3, using the detection algorithm shown in Figure 7. For a given framework and one of its client programs, AUA first checks if the classes/interfaces in the client program inherit classes/interfaces from the framework. For those that inherit, AUA further checks if the inheritance is for IOC (Inversion Of Control), then classifies those for IOC as $U1$ and the others as $U2$. Then, AUA collects the client program classes/interfaces that uses framework classes/interfaces as generic types and classifies them as $U3$. Next, AUA examines all the methods defined in client program classes/interfaces to detect $U4 - U6$, by verifying if framework classes/interfaces are used as return type, formal parameters or inside methods.

Besides identifying API usage types, AUA also checks if the used APIs are changed in the next releases of the framework by means of the outputs of the API Change Analyser. For each change, the type of API change is identified as well.

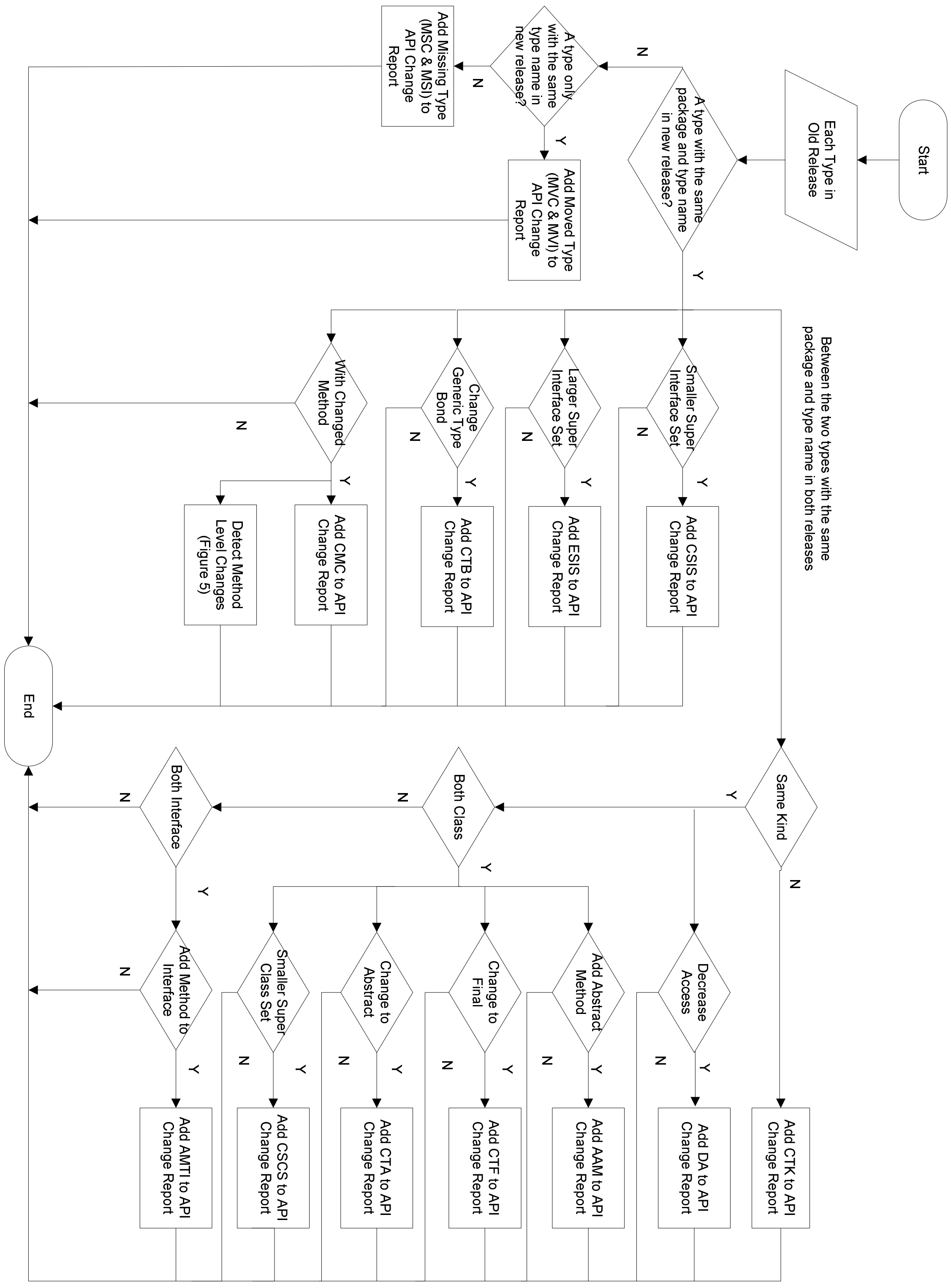

Start
Each Type in Old Release
A type with the same package and type name in new release?
N
Y
A type only with the same type name in new release?
Add Missing Type (MSC & MSI) to API Change Report
Add Moved Type (MVC & MVI) to API Change Report
Between the two types with the same package and type name in both releases
Smaller Super Interface Set
Add CSIS to API Change Report
Larger Super Interface Set
Add ESIS to API Change Report
Change Generic Type Bond
Add CTB to API Change Report
With Changed Method
Add CMC to API Change Report
Detect Method Level Changes (Figure 5)
End
Same Kind
Add CTK to API Change Report
Both Class
Both Interface
Decrease Access
Add DA to API Change Report
Add Abstract Method
Add AAM to API Change Report
Change to Final
Add CTF to API Change Report
Change to Abstract
Add CTA to API Change Report
Smaller Super Class Set
Add CSCS to API Change Report
Add Method to Interface
Add AMTI to API Change Report

Fig. 5: Type-level API change detection (Acronyms defined in Table 1)

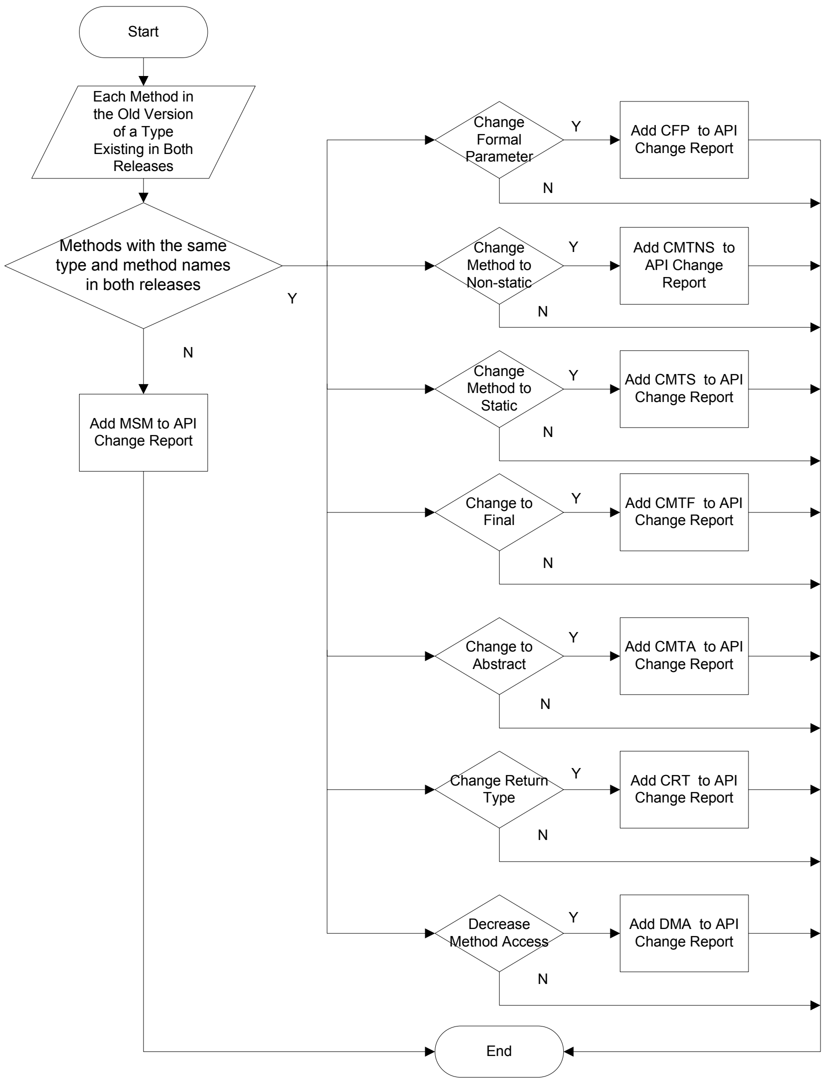

Start
Each Method in the Old Version of a Type Existing in Both Releases
Methods with the same type and method names in both releases
Y
N
Add MSM to API Change Report
Change Formal Parameter
Add CFP to API Change Report
Change Method to Non-static
Add CMTNS to API Change Report
Change Method to Static
Add CMTS to API Change Report
Change to Final
Add CMTF to API Change Report
Change to Abstract
Add CMTA to API Change Report
Change Return Type
Add CRT to API Change Report
Decrease Method Access
Add DMA to API Change Report
End

Fig. 6: Method-level API change detection (Acronyms defined in Table 2)

3.6 Running Example

To illustrate our approach, we describe the work flow of ACUA using solr-core[10] version 3.6.2 as a running example. Solr is an open source search platform based on Apache Lucene[11]. One framework used by solr-core v3.6.2 is lucene-core v3.6.2 and the Solr development team wants to upgrade to lucene-core v4.0.0 in the next release v4.0.0 of Solr. We assume that Solr developers use ACUA to collect the

[10] `http://lucene.apache.org/solr/`
[11] `http://lucene.apache.org/`

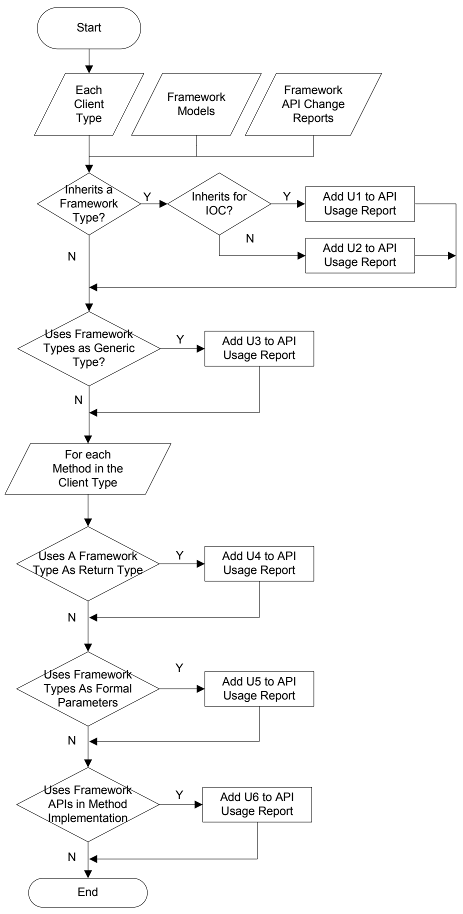

Start
Each Client Type
Framework Models
Framework API Change Reports
Inherits a Framework Type?
Y
Inherits for IOC?
Y
Add U1 to API Usage Report
N
Add U2 to API Usage Report
N
Uses Framework Types as Generic Type?
Y
Add U3 to API Usage Report
N
For each Method in the Client Type
Uses A Framework Type As Return Type
Y
Add U4 to API Usage Report
N
Uses Framework Types As Formal Parameters
Y
Add U5 to API Usage Report
N
Uses Framework APIs in Method Implementation
Y
Add U6 to API Usage Report
N
End

Fig. 7: API usage detection algorithm

information about the API changes between lucene-core v3.6.2 and v4.0.0 and how the changes are used in Solr v3.6.2, because they need to plan the upgrading task accordingly.

Solr developers would first create a POM file for v4.0.0 in which the version of lucene-core is changed to v4.0.0. A snippet of the POM file is shown in Figure 8. In this POM file, the releases of solr-core and lucene-core are represented by `<groupId>`, `<artifactId>`, and `<version>`. The release information of lucene-core is under the `<dependency>` node, while that of solr-core is at the root level.

Then, the Solr developers run ACUA with the two POM files of Solr, v3.6.2 and v4.0.0, respectively. ACUA detects the version changes in lucene-core (v3.6.2 to v4.0.0), then analyses the Jar files of the two releases of lucene-core and solr-core v3.6.2 to generate API change and API usage reports for them. To facilitate post-processing, API change and API usage reports are in XML format.

Figure 9 shows part of the API change report for lucene-core v3.6.2 to v4.0.0. This report stores the information of the two releases of frameworks, the types of

```
...
<groupId>org.apache.solr</groupId>
<artifactId>solr-core</artifactId>
<version>4.0.0</version>
...
<dependencies>
        <dependency>
                <groupId>org.apache.lucene</groupId>
                <artifactId>lucene-core</artifactId>
                <version>4.0.0</version>
        </dependency>
...
```

Fig. 8: Snippet of solr-core v4.0.0 POM file

```
<Incompatibilities>
    <programName>
    org.apache.lucene:lucene-core
    </programName>
    <oldVersion>3.6.2</oldVersion>
    <newVersion>4.0.0</newVersion>
    <incompatibilities>
        <type>AddAbstractMethod</type>
        <instances>
            <level>type</level>
            <oldAPI>
            class org.apache.lucene.analysis.Analyzer
            </oldAPI>
        </instances>
                ...
    </incompatibilities>
        ...
</Incompatibilities>
```

Fig. 9: Snippet of lucene-core v3.6.2-v4.0.0 API change report

API changes, the levels and the changed APIs in specific XML nodes. Developers can search, and analyse the API changes according to the information that they are interested in.

Figure 10 shows the API usage report, which lists the information of the client program and of the two releases of the framework to upgrade, the used APIs, the types of the usages and the change types affecting the APIs in the new release of the framework.

ACUA also provides post-process modules to convert the reports to plain text format and to summarise the contents in the reports for statistical analysis or visualisation. As shown in Section 4.2, developers can process the API change and API usage reports and visualise the distributions of the types of API changes in the frameworks, then compare them with those of the distributions of the types of API changes used in the client programs.

4 Case Study

In this section, we present the results of an empirical case study on the research questions from Section 2. First, we discuss the data set that we used.

```
<exposureReport>
    <clientName>
    org.apache.solr:solr-core
    </clientName>
    <clientVersion>3.6.2</clientVersion>
    <frameworkName>
    org.apache.lucene:lucene-core
    </frameworkName>
    <oldVersion>3.6.2</oldVersion>
    <newVersion>4.0.0</newVersion>
    <apiUsages>
        <type>EXTENSION</type>
        <apiUsages>
            <oldAPI>
            class org.apache.lucene.analysis.Analyzer
            </oldAPI>
            <invokedInFramework>
            true
            </invokedInFramework>
            <instances>
                <entities>
                class org.apache.solr.analysis.TokenizerChain
                </entities>
                <incompatibilityType>
                AddAbstractMethod
                </incompatibilityType>
            </instances>
            ...
        </apiUsages>
                ...
    </apiUsages>
        ...
</exposureReport>
```

Fig. 10: Snippet of solr-core v3.6.2 API usage report on lucene-core v3.6.2-v4.0.0

4.1 Data Set Description

We use two popular framework ecosystems as the dataset in our case study, *i.e.,* Apache[12] and Eclipse SDKs[13]. The frameworks provided by these two ecosystems are widely used in open-source software development and have evolved during the last two decades with structured dependency and version information between client programs and frameworks. Such dependency and version information is essential to study API changes and usages together.

A program can be both a client of some frameworks and a framework used by another programs [19]. In our study, if a program exists in the dependencies of other programs, we treat it as a framework and analyse its API changes. If the program also depends on other programs, we analyse the APIs that it uses as a client. It is also possible that a program is only a framework or only a client program.

Since developers of client programs inside the framework's own ecosystem ("internal client") potentially have more knowledge about the framework than developers of client programs outside the ecosystem ("third party client"), this may affect the way the developers use the frameworks and deal with API changes. Hence, we study the internal and third-party client programs of Apache and Eclipse frameworks separately. We describe how to distinguish internal and third-party client programs for the two ecosystems below.

[12] http://search.maven.org/
[13] http://marketplace.eclipse.org/

Table 5: Eclipse data set

Frameworks	Releases	# Internal Clients	# Third-party Clients
org.eclipse.core.runtime	3.1.0	47	13
org.eclipse.jdt.core	3.2.0.v_671	9	9
org.eclipse.ui.editors	3.2.0.v20060605-1400	6	3
org.eclipse.jface	3.1.0	4	1
	3.2.0.I20060605-1400	7	2
org.eclipse.team.core	2.1.0	7	1
org.eclipse.jdt.ui	2.1.0	3	1
	3.1.0	3	3
	3.2.0.v20060605-1400	4	3
	3.3.0.v20070607-0010	5	3
org.eclipse.debug.ui	3.2.0.v20060605	5	2
Distinct Clients		55	13

For the Eclipse ecosystem, we consider the Eclipse plug-ins as frameworks and client programs. An Eclipse SDK is a package of Eclipse plug-ins. Different kinds of Eclipse SDK exist, such as Eclipse SDK Classic, Eclipse SDK for Java developers, or Eclipse SDK for C/C++ developers. Eclipse SDK and Eclipse plug-ins have independent versions. For example, Eclipse SDK 3.3 may include a plug-in with the version of 3.1.

Eclipse does not have a central repository, like Maven, to host the release history of its plug-ins individually. Therefore, we downloaded 16 releases of Eclipse SDK Classic (1.0-4.3) and extracted the contained plug-in releases as the internal client programs and frameworks of Eclipse. The total number of plug-ins in the 16 releases of Eclipse SDK is 145 with 1,017 releases.

To identify Eclipse third-party client programs, we reuse previous work by Businge *et al.* [20], who studied the evolution of 21 Eclipse third-party plug-ins. However, not all of these 21 plug-ins analysed by Businge *et al.* have an explicit mapping between their versions and the Eclipse SDK versions. Consequently, we considered only the 15 plug-ins out of the 21 previously studied that have an explicit mapping to Eclipse SDK versions on their host Web sites. These 15 plug-ins account for 41 releases. We manually downloaded these third-party plug-ins and use them as third-party client programs for Eclipse frameworks.

To assess the differences between the API usages in internal and third-party client programs, we sort Eclipse framework releases by their numbers of changed APIs used by both their internal and third-party client programs. Only 11 Eclipse framework releases had changed APIs used by both internal and third-party client programs. Therefore, we use these 11 Eclipse framework releases, their 55 internal client programs (88 releases), and 13 third-party client programs (28 releases), as Eclipse data set. The names of the frameworks and the number of client programs are in Table 5. The complete list of client programs is available on-line[14].

For the Apache frameworks and their client programs, we started building our data set from a snapshot of the Maven central repository taken in November 2012, which contains more than 36,000 projects from both Apache and other providers. We distinguish the projects whose IDs start with "org.apache" as Apache frameworks or internal client programs and other projects as potential third-party client

[14] `http://www.ptidej.net/downloads/replications/emse2015-api-change-usage/`

Table 6: Apache data set

Frameworks	Releases	# Internal Clients	# Third-party Clients
org.apache.ws.commons.axiom:axiom-api	1.2.13	15	3
org.apache.lucene:lucene-core	2.0.0	21	2
	3.6.2	22	5
org.apache.maven.scm:maven-scm-api	1.4	30	2
org.apache.felix:org.apache.felix.framework	1.4.1	4	1
	3.2.2	4	2
org.apache.tika:tika-parsers	0.9	4	1
org.apache.thrift:libthrift	0.7.0	10	5
org.apache.maven.doxia:doxia-module-xhtml	1.0	6	2
org.apache.pdfbox:pdfbox	1.6.0	4	5
org.apache.tiles:tiles-servlet	2.2.2	5	1
Distinct Clients		116	29

Table 7: Summary of the whole data set

	Apache Ecosystem	Eclipse Ecosystem
# Framework Releases	11	11
# Framework Classes (All Releases)	2,930	17,469
# Internal Clients	116	55
# Internal Clients Releases	182	88
# Internal Clients Classes (All Releases)	38,092	57,181
# Third-party Clients	29	13
# Third-party Clients Releases	126	28
# Third-party Clients Classes (All Releases)	22,361	3,866

programs. We analysed the API changes and usages between the 36,000 projects. Then, following the same criteria as for Eclipse, we choose the top-11 Apache framework releases that have the most changed APIs used by both internal and third-party client programs. Consequently, the Apache data set contains 11 Apache framework releases, 116 internal client programs (182 releases), and 29 third-party client programs (126) releases. The names of the frameworks and the number of client programs are in Table 6 and the complete list of client programs is on-line[14].

A client program may use more than one framework. For example, `org.eclipse.ant.ui:3.1.0` uses both `org.eclipse.core.runtime:3.1.0` and `org.eclipse.jdt.ui:3.1.0`. Therefore, the total number of client programs is smaller than the sum of the client program counts of each framework release. A framework release can also be used by more than one release of a client program. We consider such overlapping usages of an API between releases as one usage to avoid double-counting. For example, we count the implementation of a framework interface of `org.apache.lucene:lucene-core:2.0.0` in `org.apache.jackrabbit:jackrabbit-core` 1.2.1 and 1.2.2 as one interface implementation.

Table 7 summarises the sizes of our data set and we will show in Section 6 that this is the largest of such data sets used thus far to study API change types and usages.

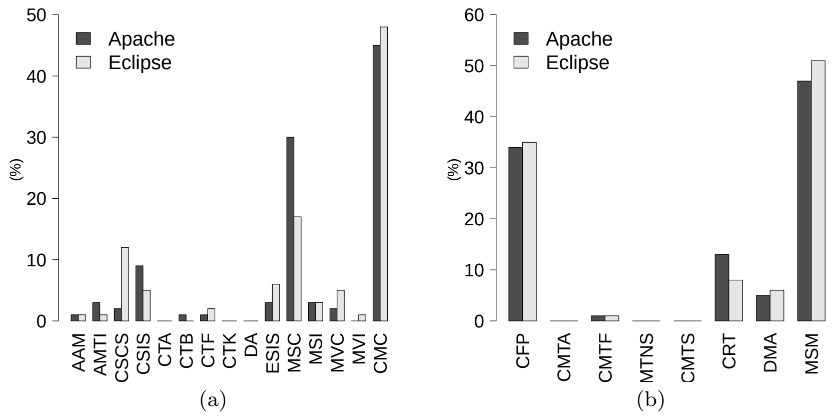


Fig. 11: P_Ts of API changes at type (a) and method (b) level

Table 8: Numbers of API changes and usages

Frameworks		Apache		Eclipse	
API		Type-level	Method-level	Type-level	Method-level
Changes		807	1,403	2,731	3,393
Usages	Internal	493 (61%)	378 (27%)	96 (4%)	43 (1%)
	Third-party	77 (10%)	58 (4%)	35 (1%)	25 (1%)

4.2 Results

For each research question, we present its motivation, the approach used, and our findings. We also discuss existing or possible approaches to help developers deal with API changes and usages, respectively.

RQ1: How often does each type of API change happen in frameworks?

Motivation: Different types of API changes may have different impacts on client programs. Knowing which types of API changes happen the most in frameworks is important for client program developers and software engineering researchers. Client program developers could take more precautions against frequent API change types, while researchers could develop approaches and tools to ease API changes that occur more often. While previous work studied API change types from different perspectives on various data sets [21–23], this study is a quasi-replication of previous studies to investigates API change types on a much larger scale, providing more generalisable results.

Approach: We compute the metric P_T, *i.e.,* the percentage of different API change types, to answer this research question.

Findings: At class/interface-level, the total number of API changes are 807 and 2,731 while there are 1,403 and 3,393 method-level API changes for Apache and Eclipse frameworks, respectively, as shown in Table 8.

The percentages in the usage rows are the ratio between the number of the usages of the APIs which are changed in the next releases of the frameworks and the total number of APIs changes in the next releases of the frameworks.

CMC (Containing Method-level Changes) and MSC (Missing Class) are the most frequent class/interface-level API changes in both Apache and Eclipse ecosystems, as shown in Figure 11(a). The CMC percentage are similar in both ecosystems, while the MSC percentage of Eclipse frameworks is about 10% lower than that of Apache frameworks. Eclipse frameworks are more likely to remove super classes of their classes (CSCS), while Apache frameworks remove super interfaces more often (CSIS). The other class/interface-level API changes occur in both ecosystems with almost the same frequency.

In the classes and interfaces with CMC, both ecosystems see similar method-level API changes, as shown in Figure 11(b). The top four types of API changes are MSM (Missing Method), CFP (Change Formal Parameter), CRT (Change Return Type), and DMA (Decrease Method Access). These changes cover 98% of the total number of method-level API changes.

Discussion MSC and MSM occur more often than the other types of API changes in the frameworks of both ecosystems. These changes may be caused by renaming or removing. Although renaming is relatively easy to adapt to, finding the correct replacements for a large number of renamed methods or classes is time-consuming. The adaptation of client programs to removed APIs is even more challenging. This finding confirms the results reported by Dig and Johnson [22], while Cossette and Walker [21] and Xing and Stroulia [23] do not report data corresponding to the API change types in their studies. However, not all changed APIs may affect the client program, which we will verify in RQ2.

If APIs still have the same reference types or method names after being changed, finding a replacement is easier for client-program developers because modern IDEs, such as Eclipse, support searching by name effectively. However, in case reference type names have changed, developers must spend considerable time identifying a suitable starting point for their upgrade, because there is no reliable way to locate the replacements of reference types with different names. Although IDEs support fuzzy searches and there exist approaches to generate API change rules, neither are 100% accurate [13,14,24]. In such cases, developers must find the replacements of the missing classes, interfaces, and methods manually.

For unavoidable API name changes, effective documentation can ease the upgrading process. Especially more detailed descriptions about where and when the new APIs are used in the new releases of frameworks is necessary for inheritance-style usage. Although it is time-consuming to provide detailed documentations for frameworks, even imperfect change rules generated by tools [13,14,24–26] could already help developers find a replacement for missing APIs and could be used as supplement to insufficient documentation, as shown by a recent empirical study [27]. An alternative source for documentation could be generated during code-refactoring process, as Dig *et al.* [22] showed that more than 80% of API changes are caused by refactoring.

The differences between the numbers of changed APIs between Apache and Eclipse frameworks may be due to our choice of considering different *versions* of the Eclipse SDK while Apache frameworks do not follow the same versioning principle.

RQ2: How often does each type of API change affect client programs?

Motivation: Knowing how APIs change in frameworks, we investigate if these changed APIs affect client programs. If the answer is no, it would be a proof that framework developers avoid changing APIs used by client programs. If the answer is yes, framework developers might not be able to ease client program adaptation by keeping APIs stable all the time and client-program developers should be careful in isolating APIs in their programs (through the use of Faades, for example) to minimise the impact of the API changes.

Therefore, on the one hand, framework developers should provide special documentations to help client-program developers to upgrade their programs whenever API changes occur during a framework evolution (not only documentation about the new APIs) while, on the other hand, client-program developers should isolate API usages using language mechanism and design strategies at their disposal: method wrappers, delegates, and the Abstract Factory and Faade design patterns.

Researchers could also develop techniques and tools to ease the upgrading process of client programs to adapt to such frequent changes. There was no answer to this question base on large-scale analyses of API change and usage together yet.

Approach: We cross-reference API usages with API change types and calculate P_{T_U}, *i.e.,* the percentage of a specific API change type affecting a client program, and $P_{T\_M\_U}$, *i.e.,* P_{T_U} restricted to methods, to answer this research question.

Findings: CMC (Containing Method-level Changes) and MSC (Missing Classes) again affect client programs more often than the other API changes, as shown in Figures 12(a). The influence of these two types of API changes is similar on internal client programs while CMC affects third-party client programs more often than MSC.

MSI (Missing Interfaces) affects Eclipse internal client programs substantially, yet does not occur often in frameworks (17% of P_{MSI_U} vs. 3% of P_{MSI}) as shown in Figure 12(a). Furthermore, when taking the used methods of the interfaces affected by MSI into account, MSI again affects Apache internal client programs substantially, yet does not occur often in Apache frameworks as well (24% of $P_{MSI\_M\_U}$ vs. 3% of P_{MSI}) as shown in Figure 12(b).

The method-level API changes in the classes/interfaces under CMC affect client programs almost to a same degree as they happen in frameworks with two main differences. The first is that DMA (Decrease Method Access) rarely affects client programs. The second is that Apache client programs are more often affected by CFP (Change Formal Parameters) while Eclipse client programs are more often affected by MSM (Missing Method).

The numbers of API changes affecting client programs are much lower than those actually happening in frameworks, as shown in Table 9. Hence, despite of the large number of API changes in Eclipse frameworks, few of them affect client programs, which is good news for both client and framework developers.

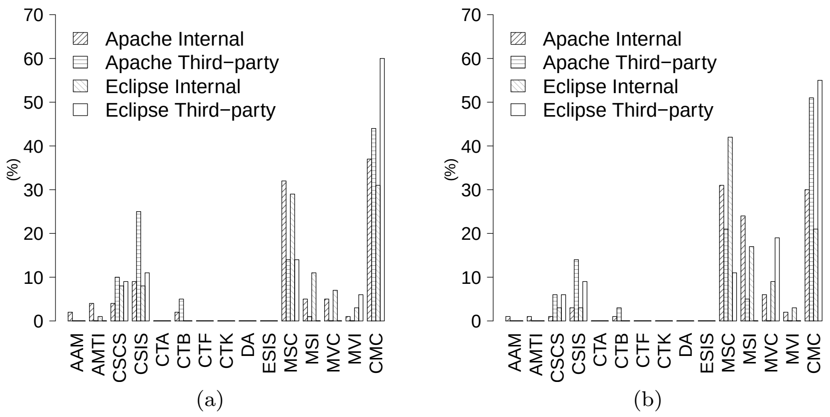


Fig. 12: Type-level API changes affecting client programs ((a) P_{T_U} and (b) $P_{T\_M\_U}$)

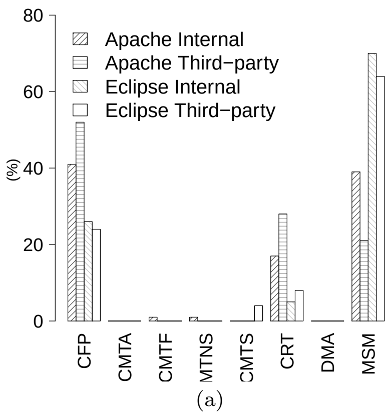


Fig. 13: Method-level API changes used in client programs (P_{T_U})

Discussion Similar to what we observed in API changes, CMC and MSC also affect client programs more often. This finding shows that framework developers should be concerned about the impact of those API changes and adopt countermeasures such as more detailed upgrading documents. As mentioned earlier, client program developers could use tools [13, 24–26, 14] that build API change rules to ease the upgrading process when documentation is not available.

We also noticed that MSI affects internal client programs often, which may be caused by internal client programs more likely to integrate with frameworks through IOC.

Eclipse Provisional API Guidelines distinguish official and internal APIs. This guideline may contribute to the observation that less API changes in Eclipse

Table 9: Number of type-level API changes affecting client programs ($\#T\_U$ and $\#T\_M\_U$)

Framework		Apache		Eclipse	
API		$\#T\_U$	$\#T\_M\_U$	$\#T\_U$	$\#T\_M\_U$
Usage	Internal	493	1,453	96	232
	Third-party	77	137	35	47

frameworks affect client programs. On the one hand, Eclipse framework developers change official APIs less often. On the other hand, client program developers use internal APIs less often as well.

Furthermore, we observe that, overall, only a small percentage of APIs used in client programs were changed during framework evolution in our study. This observation may be due to us considering only API changes between consecutive releases of frameworks. While, we would expect that, in general, only a small number of APIs change between consecutive releases, occasionally, a large number of changes can happen between consecutive releases, as shown in Figure 14. For example, the large number of API changes between v3.2.2 and 4.0.0 is due to implementing many features of the then newly-released OSGi v4.3, its underlying framework[15]. Moreover, in practice, the version gap between framework releases can be larger than what we observed in this work because developers may have to upgrade multiple frameworks at the same time, as shown by Raemaekers *et al.* [1], which would make the problem of upgrading framework APIs even more critical.

RQ3: How widely are APIs used in the client programs?

Motivation: Like any other software artefact, frameworks are evolved very often and client programs must keep upgrading in order to use new functions or patch security vulnerabilities. Knowing how widely APIs are used in client programs will help developers estimate the impact of API changes. For example, a changed API used in many locations in client programs may cause the Shotgun Surgery [28] code smell, *i.e.,* many little changes in multiple classes, which would result in high upgrade costs.

Approach: We use two metrics to answer this research question: USR (USage Ratio) and IUSR (Ideal USage Ratio). The former represents the current status of API usages while the latter reflects the minimum achievable API usages. We use the Mann-Whitney-Wilcoxon test [29] to verify if the USRs and IUSRs between Apache and Eclipse internal and third-party client programs are statistically significantly different. The Mann-Whitney test is non-parametric and does not require the knowledge of the distributions of the data. Besides Mann-Whitney-Wilcoxon test, we also show the results in a beanplot that visualises the density of the data.

[15] `http://svn.apache.org/repos/asf/felix/releases/org.apache.felix.framework-4.0.0/doc/changelog.txt`

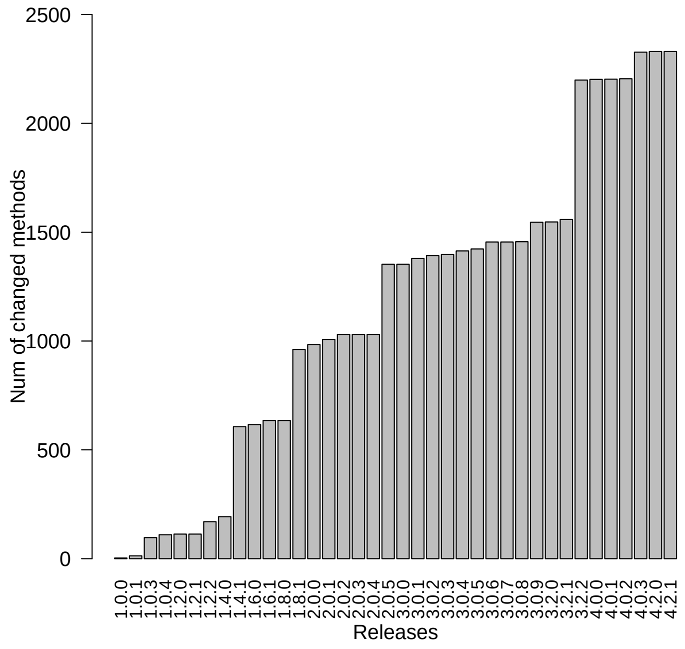


Fig. 14: Accumulated number of changed methods of org.apache.felix.framework

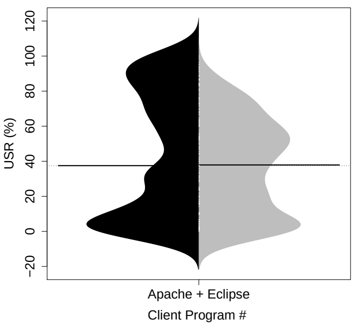
Table 10: Amount of inheritance- and composition-usages

	Apache		Eclipse		Total
Client Programs	Internal	Third-party	Internal	Third-party	
Inheritance	3430	193	2848	145	6616
Composition	2040	522	8995	583	12100

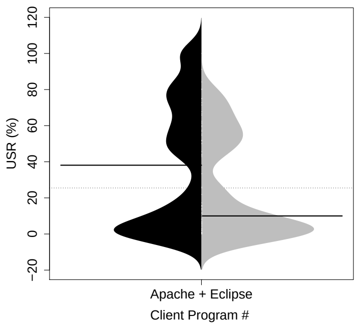
Findings: The medians of the USRs are 38% and 32% in Apache and Eclipse client programs, respectively. However, the median of the IUSRs are 0% and 1% for the client programs of the two ecosystems, respectively. Almost all USRs and IUSRs between internal and third-party client programs are statistically different as shown in Table 11 and in Figure 15, with the USRs/IUSRs of third-party client programs being smaller than those of internal client programs. Although the difference of the USRs between Apache internal and third-party client programs are not significant, the p-value is still low (0.06).

The USRs of third-party programs are statistically different between Apache and Eclipse ecosystems, with those of Eclipse being lower than those of Apache. However, the other USR/IUSR differences between internal client programs of the two ecosystems are not significant.

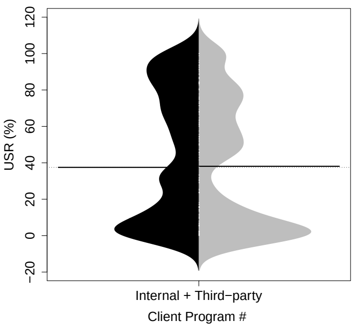
Discussion In most cases, the USRs and IUSRs of internal client programs are higher than those of third-party client programs, but both can reach 100%, which



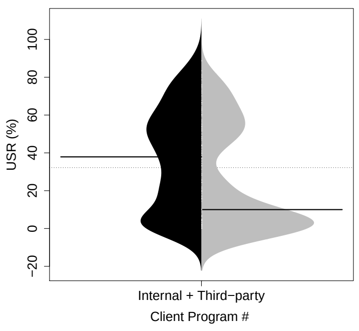
(a) Apache vs. Eclipse internal



(b) Apache vs. Eclipse third-party



(c) Apache internal vs. third-party



(d) Eclipse internal vs. third-party

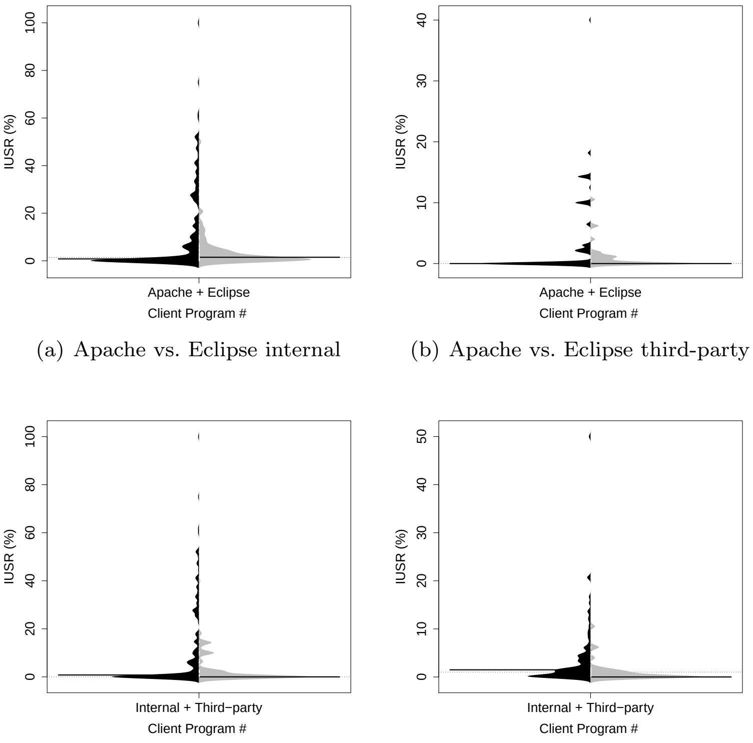
Fig. 15: Usage Ratios (USRs)

would mean that all classes and interfaces of these client programs could be affected by API changes. Widely-spread API usages are more difficult to adapt to API changes. Client developers should keep monitoring the USRs of their client programs and keep their values as low as possible. The difference between USR and IUSR shows the room for reducing API usages.

IUSR is the lower boundary of USR, with the ratio between inheritance- and composition-usages is about 1 to 2. The difference between IUSRs and USRs shows that composition-usages can replace most of the inheritance usages that are not used for IOC. However, there might be additional costs to reach the minimum value of USRs, *i.e.,* the ideal usage ratios (IUSRs), for all the frameworks, because composition-style usages require extra effort of client program developers to encapsulate framework APIs instead of using them directly. Developers may decide on how tight the coupling between their programs and a framework should be, according to the stability of the frameworks or resource constraints, such as development time period. For stable frameworks, developers could consider coping with the current USR level as those APIs will rarely change between releases. Instead, client developers should reduce USRs for unstable or obsolete frameworks, as this will probably minimise the changes required to upgrade to a new release.

Table 11: Mann-Whitney-Wilcoxon test results on USR and IUSR

p-value	Apache vs. Eclipse		Internal vs. Third-party	
	Internal	Third-party	Apache	Eclipse
USR	0.11	$< \mathbf{0.05}$	0.06	$< \mathbf{0.05}$
IUSR	0.70	0.44	$< \mathbf{0.05}$	$< \mathbf{0.05}$



(a) Apache vs. Eclipse internal (b) Apache vs. Eclipse third-party

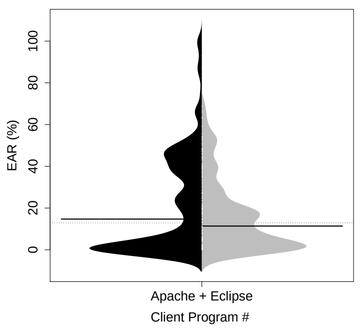
(c) Apache internal vs. third-party (d) Eclipse internal vs. third-party

Fig. 16: Ideal Usage Ratios (IUSRs)

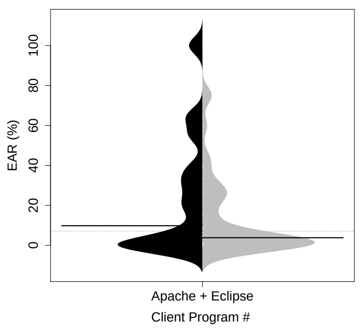
RQ4: What percentage of framework API usages are the API-injection usages?

Motivation: We found that, on average, there is more than 30% difference between USR and IUSR in client programs of the frameworks in the two ecosystems. As discussed in Section 2, the difference can be caused and amplified by $U2 - U5$ in Table 3. It is important to know how much of the API usages belong to $U2 - U5$, since avoiding such usages could be an efficient way to reduce USRs of client programs.

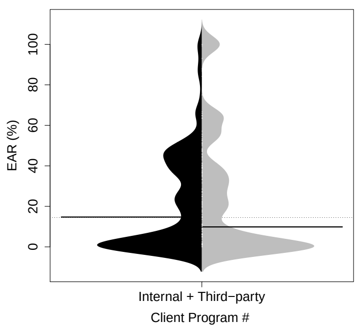
Approach: We use EAR (Encapsulable API-Injection Ratio), the proportion of $U2-U5$ in total API usage, to answer this research question. We also use the Mann-



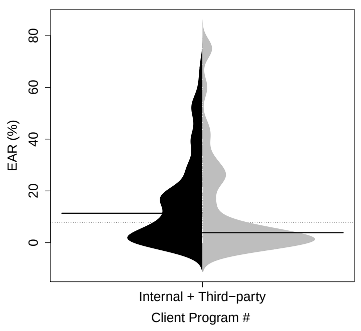
(a) Apache vs. Eclipse internal



(b) Apache vs. Eclipse third-party



(c) Apache internal vs. third-party



(d) Eclipse internal vs. third-party

Fig. 17: Encapsulable API-Injection Ratios (EARs)

Table 12: Mann-Whitney-Wilcoxon test results on EAR

p-value	Apache vs. Eclipse		Internal vs. Third-party	
	Internal	Third-party	Apache	Eclipse
EAR	0.58	0.32	0.66	0.07

Whitney-Wilcoxon test [29] to verify if the EARs between Apache and Eclipse internal and third-party client programs are statistically different.

Findings: The values of EARs are not statistically different between Apache and Eclipse client programs, with medians of EARs of 14% and 8%, respectively, as shown in Table 12 and in Figure 17.

Discussion As recommended by Bloch [12], composition-style usage can encapsulate framework APIs and prevent propagating their changes. Also, composition-style usage avoids the *fragile base class problem* [30] affecting client programs, while internal changes in base classes could break client programs, even if there is no API change. Approaches and tools that help developers for such specific encapsulation tasks are interesting for researchers in software engineering.

Inheritance-style usage or $U1$ also is not avoidable, because many frameworks are designed for IOC. However, developers can still protect client programs by loosening the coupling to framework APIs: The Facade or Adapter patterns [2] can provide a buffer layer between client programs and frameworks, enabling client programs and frameworks to evolve relatively independently.

The reduction of framework API usage in client programs can then be illustrated in Figure 18. Ideally, client programs should only keep $U1$ and $U6$ and encapsulate $U2-U5$ with composition-style usage. Tools like ACUA [10] can help developers by identifying $U2-U5$ usages. Modern IDEs, such as Eclipse, also provide refactoring tools, for example to extract methods from other pieces of code. These refactoring tools would help developers apply series of refactorings to extract code that uses framework APIs into one single class. For example, the class `org.eclipse.swt.internal.win32.OS` of Eclipse[16] is a class encapsulating all the API calls from SWT to the underlying Win32 platform.

Developing automated refactoring tools for the reduction of API usage is an open problem [31]. However, Shah *et al.* [32] demonstrated that Introduce Factory and Move refactorings provided by the Eclipse IDE can be used to reduce the dependencies in Java programs automatically. Their work is an example of the feasibility of using automated tools to reduce $U2-U5$ API usages.

RQ5: Which type of API change affects the encapsulable API-injection usages more often?

Motivation: API-injection usages, *i.e.,* $U2-U5$, can propagate framework API changes to other parts of the client programs. However, the effort of client program developers to adapt to API changes depends on the API change types. We study the types of API changes that affect the injected client program APIs to understand their prevalence and potential impact.

Approach: We first verify if the API changes and $U2-U5$ are correlated with Pearson's Chi-squared and Fisher's exact tests. If the answer is positive, we further investigate which types of API changes affect $U2-U5$ more often than the others with Odds Ratio of the different types of API changes.

We use CMC (Containing Method-level Changes) as the reference group to compute the Odds Ratio, because CMC is the most frequent API change. The Odds Ratio of CMC is one, the API change types with Odds Ratios greater than one are considered more common while those with Odds Ratios less than one are less common.

Findings: The statistical test results in Table 13 show that, for Apache internal clients, only encapsulable API-injection usages ($U2-U5$) are correlated to API changes. As shown in Table 14, CMC, MSC (Missing Classes), and CSIS (Contract SuperInterface Set) have the largest numbers of API-injection usages. AAM (Add Abstract Method), MVI (Moved Interface), and CTK (Change Type Kind) are the three API change types that are always used in $U2-U5$.

[16] `http://www.docjar.com/docs/api/org/eclipse/swt/internal/win32/OS.html`

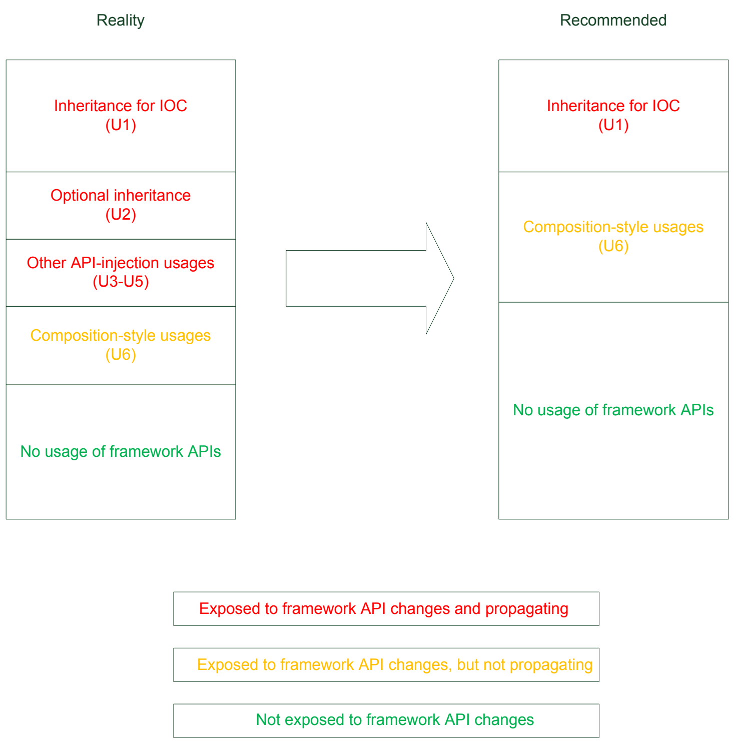

Reality
Recommended
Inheritance for IOC (U1)
Optional inheritance (U2)
Other API-injection usages (U3-U5)
Composition-style usages (U6)
No usage of framework APIs
Inheritance for IOC (U1)
Composition-style usages (U6)
No usage of framework APIs
Exposed to framework API changes and propagating
Exposed to framework API changes, but not propagating
Not exposed to framework API changes

Fig. 18: API usage recommendation

Table 13: Correlations between encapsulable API-injection usages

p-value	Apache		Eclipse	
	Internal	Third-party	Internal	Third-party
Chi-Square	$<$ **0.05**	0.52	0.20	0.12
Fisher's Exact	$<$ **0.05**	0.62	0.21	0.43

Table 14: Odds ratio of API change types in Apache internal client programs

API change types	$U2-U5$ usages	$U1$ and $U6$ usages	Odds ratio
AAM	10	0	Inf
MVI	3	0	Inf
CTK	2	0	Inf
CSIS	26	5	1.47
CMC	60	17	1.00
MVC	7	2	0.99
CSCS	7	2	0.99
CTN	4	2	0.57
AMI	11	7	0.45
MSC	29	23	0.36
MSI	8	15	0.15

Discussion The correlation between encapsulable API-injection usages in Apache internal client programs may be caused by the fact that Apache does not have an explicit policy to regulate API changes, while Eclipse has Provisional API Guidelines. Hence, Apache framework developers have more freedom to change APIs than Eclipse framework developers. Being injected in the client program APIs may be a factor causing CMC and MSC to affect client programs more frequently. Although AAM, MVI, and CTK are always used in $U2 - U5$, these three API change types are easier to adapt than MSC and MSM, which are the main causes of CMC.

4.3 Summary

The goal of our study is to better understand how APIs are changed and used on a large scale, so both framework and client program developers can mitigate the impact of API changes pro-actively. Analysis of the top-11 framework releases from Apache and Eclipse SDKs with the most changed APIs affecting their internal and third party client programs, showed that:

– Framework developers should be more disciplined to avoid removing or changing the names of reference types and methods used as APIs. We found that Missing classes (MSC) and methods (MSM) are the most frequent API changes and they affect client programs more often. Comparing to other API changes, such as removing method parameters, these two types of API changes require developers to search for replacements or to reimplement the changed APIs. For unavoidable API name changes, framework developers should provide detailed documentation to guide client program developers to upgrade, with the help of API change rule detection tools.
– Framework developers should have clear policies about which APIs are accessible and stable. For example, Eclipse Provisional API Guidelines distinguish between official and internal APIs. This distinction explains why API changes in Eclipse frameworks affect less client programs than those in Apache frameworks. As shown in Table 8, Eclipse frameworks have more API changes than Apache frameworks, but less API changes affecting client programs than Apache frameworks.
– A common solution for client developers to contain API usage within isolated parts of their code, is to use composition, as changes to widely-spread API usages are more difficult to adapt to. We found that the median of the ideal usage ratios (IUSRs), which represent the API usages for inversion of control (IOC), are 0% and 1% for the client programs of Apache and Eclipse client programs, while the medians of USRs for all API usages, are 38% and 32% for the client programs of the two ecosystems, respectively. The differences between IUSRs and USRs show that client program developers can encapsulate their unnecessary API usages (*i.e.,* usages not performed because of IOC) with composition to reduce USRs. However, there might be additional cost to reach the lower boundary of USRs, *i.e.,* the ideal usage ratios (IUSRs), because composition-style usages require the extra effort of client program developers to encapsulate framework APIs instead of using them directly. Developers should decide on how tight the coupling between their programs and a framework should be,

according to the stability of the frameworks and resource constraints, such as the development time period.

5 Threats to Validity

Our study, its results, and our conclusions are subject to several threats to their validity [29], which we discuss below.

Construct Validity verifies that the observations really reflect the theory, *i.e.,* whether the treatment reflects the cause and the outcome reflects the effects. We wanted to observe whether different API change types occur in frameworks and affect client programs differently, depending on their usages of the APIs. Therefore, we chose one categorisation of API changes and usages among different possibilities. As the basis of our analysis, we followed a categorisation proposed by experienced practitioners[2].

Internal Validity verifies that the outcome is really caused by the treatment. There may be errors in the tool used to collect data in our study or the tool may overlook some of the categories of changes types and usages. We carefully tested our tool, ACUA, and wrote several unit and regression tests. During data analysis, we did not observe inconsistent or conflicting results. We also described ACUA algorithms in detail to help with future replication by other researchers. Finally, our implementation of ACUA is available on-line[14].

ACUA does not analyse API usages via Java reflection because this would require inferring the types and methods described by Strings either through sophisticated, and costly, static analyses or through incomplete dynamic analyses. However, a search for uses of the Java reflection API convinced us that such cases are very rare with respect to the number of non-reflective API calls, most likely for performance reasons. Therefore, this limitation does not have an observable effect on our results.

In the API change detection algorithm, we consider that a class (or an interface, respectively) is moved to another package in the new release of a framework if (1) the class does not exist in the same package in the new release and (2) a class with the same simple name is added to another package in the new release. Since the deleted and added classes can have the same name but irrelevant, ACUA could generate false positive class/interface moves. We randomly sampled 10% of about 174 class/interface moves and did not find any false positive.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details about our study to help others to replicate it. In particular, all studied programs are publicly available. ACUA and the raw data used in this study are freely available on-line[14].

External Validity verifies that the results of a study are generalisable. We only analysed Java frameworks of Apache and Eclipse. Although popular and large, these frameworks are not representative of the general population of frameworks in Java and in other programming languages.

Because Eclipse does not have a central repository, like the Maven central repository for Apache, and because we cannot know the general population of all client programs (even with a central repository), all the results and conclusions of our study are only valid in the context of our data set. Yet, we performed to date the largest study on API changes and usages and, as such, our results and conclusion can be helpful to practitioners and researchers.

Finally, many client programs are also frameworks. The lack of clear difference between API usages in frameworks and "pure" client programs may affect our results. Future work should distinguish frameworks from "pure" client programs and assess their differences in terms of API changes and usages.

6 Related Work

Researchers studied framework APIs from different perspectives: change mapping [33,24,26,14], documentation [34–36], teaching/learning [37–39], usage examples [40,41], refactoring [42,43], stability measurement [44,1], migration [45,46], Web API evolution [47]. In the following, we present work related to API changes, API usages, and the combination of both. We also compare ACUA to existing tools and summarise the difference between our work and previous works and tools, as shown in Table 16 and Table 17.

6.1 API Changes

Des Rivières[2] discussed in detail API contract compatibility and classified API changes according to the elements of the Java APIs, such as package, class, method, and so on. He did not investigate which API change types occur in frameworks and client programs. We base the categorisation of API change types in ACUA on this previous work.

Raemaekers *et al.* [48] investigated whether the developers of the frameworks in the Maven central repository follow the semantic versioning rule, which suggests that the version format be MAJOR.MINOR.PATCH. According to this version format, only MAJOR versions should introduce incompatible API changes. Yet, they found that the semantic versioning rule is not enforced in the frameworks hosted in the Maven central repository and that MINOR and PATCH versions of some frameworks introduced incompatible API changes. We drew inspiration from this previous work to study the distribution of API changes in frameworks and their impact on API usages in client programs.

Hou and Yao [3] classified API changes in AWT and Swing according to domains and design intentions. They identified eight design intentions for the API changes in these two JDK packages, *e.g.,* changes in naming conventions or changes to introduce a new concept. They found that the proportion of changed APIs is small but feature redesign is an important cause of API changes. They did not study the impact of the API change types on client programs using AWT or Swing.

Cossette and Walker [21] investigated manually the binary incompatibility between 16 pairs of releases of three Java systems (Struts, Log4j, an JDOM). According to the easiness to adapt to API changes, they classified API changes into three categories: fully automatable, partially automatable, and hard to automate.

They reported the purposes of API changes from a different perspective than that of [3], *e.g.,* exposing internal implementation or generalizing. Although detailed and insightful, manual investigation is hard to apply on larger scale.

Other works, such as the work by Dig *et al.* [15] and Xing and Stroulia [23] also discussed API change types. Yet, these works considered more coarse-grained API change types than the ones in the work by Des Rivières [2], on which we base ACUA. Also, they do not study or report the distributions of API change types in between framework releases and their impact on API usages.

There is a large body of work on identifying changes between two versions of a programs or framework. For example, Xing and Stroulia [49] proposed an approach to match methods between two versions of a framework to help developer understand the changes between the two versions and, ultimately, to ease the upgrading of their client programs. Many approaches exist [13, 24, 50, 14], but these are not directly related to the analysis of API changes and usages.

6.2 API Usages

Lämmel *et al.* [7] conducted a large-scale AST-based analysis on framework-style and library-style API usages in the Ant Java projects hosted in the SourceForge Repository[17]. They were the first to study API usages on a large-scale. They observed that the number of APIs grows with framework sizes, that less than half of the APIs are used by client programs, and that half of the frameworks are used in "framework-style" (Inversion of Control). Our study of API changes and usages is inspired by and extends this work, although we parsed Java class-files instead of source code because JAR files are usually readily available and, thus, we can study a larger data set. We analysed Maven projects instead of Ant projects because, thanks to Maven central repository, we could download both frameworks and their client programs and also identify all dependencies. We fully automated the fact-extraction process (the tool is available on-line[14]), investigated API changes types and usages together, and studied the encapsulation of APIs to avoid change-propagations.

De Roover *et al.* [8] conducted a multi-perspective analysis of API usages in the Qualitas corpus [5]. They first extended this corpus with information about the dependencies, API names, domains, and facets (groups of sub-functions of frameworks) and named this extension the QuaAtlas (Qualitas API Atlas). They then explored API usages in QuaAtlas from different perspectives and presented insights on these usages. They also proposed a Web-based tool, Exapus, for developers to explore API usages. In our study, we relate API usages with API changes and focus on upgrading effort.

Other researchers studied API usages from different points of view. The tool SpotWeb developed by Thummalapenta and Xie [39] can identify frequently-used and rarely-used framework APIs by mining open-source software repositories. It does not directly allow understanding the impact of APIs changes on their usages.

Kawrykow and Robillard [40] proposed an approach to detect redundant code, *i.e.,* pieces of code in client programs that imitate implementations of APIs in frameworks and, thus, that could be replaced by API calls. Their approach cannot

[17] `http://sourceforge.net/`

be used readily to compute the distributions of API changes and usages, and the impact of the change types, across releases of frameworks.

LibSync [51] helps developers learn complex APIs and adapt their client programs to changes by mining usage change patterns in client programs that have been upgraded already. It is useful for developers but does not provide information about the types of changes and their impact on API usages. Also, it does not provide advice to limit change propagations. Another technique, APIMiner [52] helps developers by extracting examples of API usages using program slicing.

Businge *et al.* [6] studied 512 Eclipse Third-Party Plug-ins (ETPs) and their usages of Eclipse internal and official APIs. They observed that 44% of these ETPs use internal Eclipse APIs and that ETP developers continue to use these internal APIs even when evolving their plug-ins. They thus showed that API usages may have an important impact on upgrading costs but did not relate the observed usages with change types.

6.3 API Changes and Usages

Few previous work studied API changes and usages together. Dietrich *et al.* [4] investigated the differences between Java compile-time and link-time compatibility and their influences on client programs. They analysed 564 releases of 109 programs from the Qualitas corpus [5], excluding those from Eclipse and Azureus. They found that 75% of upgrades have link-time incompatibilities. Some incompatibilities also exist at compile-time but only affect eight client programs. They did not consider finer-grained change types and did not recommend measures to minimise upgrading costs.

Robbes *et al.* [9] conducted a study on API deprecation in the Smalltalk Squeak and Pharo ecosystems and observed its impact on developers. They analysed 577 and 186 deprecated methods and classes, respectively, and discovered that 14% of the deprecated methods and 7% of the deprecated classes caused changes in at least one project. The median of the developers' reaction time to changes was two weeks. Yet, they also observed that only about 20% of affected projects reacted to deprecated APIs. They did not consider other type of changes, besides deprecation, and studied a less popular language, while we consider a much larger data set by considering frameworks and client programs in Java.

6.4 Other Tools

Besides research approaches, there are also other tools from the open-source software community analysing API change or usage in Java programs. The differences between these tools and ACUA are summarised in Table 15.

Animal Sniffer[18] verifies if a program is compatible with the APIs of dependent frameworks. It analyses both API changes and usages, but only checks which APIs are changed or used, not how (much) they are changed or used. Similar to Animal Sniffer, Clirr[19] tries to identify both binary and source incompatible API

[18] `http://mojo.codehaus.org/animal-sniffer/`
[19] `http://mojo.codehaus.org/clirr-maven-plugin/index.html`

Table 15: Tool comparison

	API			
	Changes	Change Types	Usages	Usage Types
Animal Sniffer	Yes	No	Yes	No
Clirr	Yes	Yes	No	No
Java ACC	Yes	Yes	No	No
JAPITools	Yes	NA	No	No
JBoss Tattletale	No	No	Class	No
JDiff	Yes	Yes	No	No
ACUA	Yes	Yes	Yes	Yes

changes between two releases of a framework. It does not analyse API usages. Java API Compliance Checker (Java ACC)[20] is a Java API backward-compatibility verification tool. Similarly to ACUA, it is based on the API change classification of Des Rivières [2] but it does not identify API usages. JDiff[21] and JAPITools[22] are two tools similar to Java ACC, but they detect more coarse-grained API change types. Tattletale[23] is a tool developed by JBoss to analyse JAR, WAR, and EAR file dependencies in Java projects to minimise the number of dependencies and eliminate duplicated class files. Tattletale works mainly at class level and does not analyse API usages in details nor does it consider API changes.

6.5 Summary

We drew inspiration from this previous work and bring the following contributions. We conducted an automated analysis on both API changes and usages in the frameworks and client programs of two popular framework ecosystems: Apache and Eclipse. Table 7 shows that the presented results pertain to 22 frameworks, 213 client programs, 424 releases of these programs, for a total of 141,899 classes. These frameworks and client programs are selected by analysing 36,000 projects from Maven central repository and 16 releases of Eclipse SDK classic, which make it the largest study on the topic of APIs changes to date.

We grouped the API changes according to Des Rivières' classification and investigated how these API change types affect client programs depending on the usages of the APIs by these programs.

We confirmed the findings in Raemaekers *et al.* [48] and Dietrich *et al.* [4] that developers remove classes and methods very often. Because other API change types are different between ours and the previous studies or the other previous studies do not report the distribution of API change types, we cannot compare our results with theirs. We also identified the client program classes or interfaces that are currently exposed to and can be protected from API changes. Our study contribute to provide more empirical data about the reality of API changes and usages. Thus, future work could build on this work to propose automated refactorings to protect

[20] `http://ispras.linuxbase.org/index.php/Java\_API\_Compliance\_Checker`
[21] `http://jdiff.org/`
[22] `http://www.sab39.org/Software/Japitools/42/`
[23] `http://tattletale.jboss.org/`

client programs from change propagation, but also could study further types of changes and their impact on API usages in these and other ecosystems.

7 Conclusion

When frameworks evolve, the APIs that they provide may change between releases. Upgrading to new releases of frameworks is costly for developers of client programs, but delaying upgrading prevents developers from benefiting bug fixes, new features, and security patches. Differences in API usages affect upgrade costs differently.

We follow Des Rivières' definitions [2] of API change types and consider public and protected Java software entities, such as classes and methods, as APIs. We also consider five types of API usages, which we call API-injection, because they can propagate API changes within client programs and increase API usages, *i.e.,* the framework APIs are *injected* in the client program APIs.

Based on these definitions, we develop a tool to analyse API changes and usages *together*, ACUA. Using ACUA, we conduct an exploratory study on API changes and usages in the frameworks of the Apache and Eclipse ecosystems and some of their client programs. We build a data set composed of 11 framework releases from Apache and Eclipse, which represent 20,399 classes, 171 internal client programs, which include 95,273 classes, and 42 third-party client programs, composed of 26,227 classes.

In this data set, we observe the following facts. (1) Missing classes and methods happen most often in new framework releases and affect client programs more often as well than the other API changes. Missing Interfaces are rare in frameworks but affect client programs often. The API change types at method level occur in new framework release and affect the client programs almost with the same frequency. (2) On average, the APIs of a framework are used in 35% of the classes and interfaces of a client program. (3) We also observe that most of these usages could be reduced through refactorings. (4) About 14% and 8% of APIs in Apache and Eclipse frameworks are injected into client program APIs and can be encapsulated with client program APIs. Such encapsulation seems to be the most efficient way to reduce general framework API usage in client programs. (5) API changes and encapsulable API-injection usages are correlated only in Apache internal programs. This correlation may be caused by Apache not having explicit policy to regulate API changes.

Based on our observations, we suggest that client-program developers use tools such as ACUA to analyse their framework API usages regularly to plan proactive framework upgrades and to apply patterns like the Adapter or Facade design patterns to control API change-propagation. For framework developers, we suggest to avoid API changes that occur in client programs more often than others and that are more difficult to accommodate for client-program developers. Alternatively, framework developers should provide more detailed documentations regarding the API changes to help client-program developers. Empirical studies to quantitatively evaluate and exploit the benefits of ACUA in supporting framework upgrading are future work.

Acknowledgements

We thank Daniel German for providing the Maven central repository snapshots. This work has been partly funded by FQRNT, NSERC, and the Canada Research Chair on Patterns in Mixed-language Systems. We thank the anonymous reviewers for their invaluable comments on previous versions of this paper.

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. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
3. D. Hou and X. Yao, "Exploring the intent behind api evolution: A case study," in *WCRE*, 2011, pp. 131–140.
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. E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336–345.
6. J. Businge, A. Serebrenik, and M. van den Brand, "Eclipse api usage: the good and the bad," *Software Quality Journal*, pp. 1–35, 2013.
7. R. Lämmel, E. Pek, and J. Starek, "Large-scale, ast-based api-usage analysis of open-source java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11. New York, NY, USA: ACM, 2011, pp. 1317–1324.
8. C. D. Roover, R. Lämmel, and E. Pek, "Multi-dimensional exploration of api usage," in *ICPC*, 2013, pp. 152–161.
9. R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation?: The case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 56:1–56:11.
10. W. Wu, B. Adams, Y.-G. Guéhéneuc, and G. Antoniol, "Acua: Api change and usage auditor," in *4th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, ser. SCAM'14, 2014.
11. N. Ali, Z. Sharafi, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study on requirements traceability using eye-tracking," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM 2012, 2012.
12. J. Bloch, *Effective Java (2nd Edition) (The Java Series)*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008.
13. 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.
14. 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.
15. D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming.* Springer Berlin / Heidelberg, July 2006.
16. K. Pearson, "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling," *Philosophical Magazine Series 5*, vol. 50, no. 302, pp. 157–175, 1922.
17. R. A. Fisher, "On the interpretation of χ^2 from contingency tables, and the calculation of p," *Journal of the Royal Statistical Society*, vol. 85, no. 1, pp. 87–94, Jan. 1922.

18. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.
19. G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: The case of apache," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 280–289.
20. J. Businge, A. Serebrenik, and M. van den Brand, "An empirical study of the evolution of eclipse third-party plug-ins," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: ACM, 2010, pp. 63–72.
21. B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 55:1–55:11.
22. D. Dig and R. Johnson, "How do apis evolve? a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, 2006.
23. Z. Xing and E. Stroulia, "API-evolution support with diff-CatchUp," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818 – 836, December 2007.
24. M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 333–343.
25. 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.
26. 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.
27. W. Wu, A. Serveaux, Y.-G. Guéhéneuc, and G. Antoniol, "The impact of imperfect change rules on framework api evolution identi
cation:an empirical study," *Empirical Software Engineering*, vol. In Press, 2014.
28. M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
29. C. Wohlin, P. Runeson, and M. Höst, *Experimentation in Software Engineering: An Introduction.* Springer, 1999.
30. L. Mikhajlov and E. Sekerinski, "A study of the fragile base class problem." in *IN EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING*, ser. ECOOP'98. Springer, 1998, pp. 355–382.
31. M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 1037–1039.
32. S. M. A. Shah, J. Dietrich, and C. McCartin, "On the automation of dependency-breaking refactorings in java," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 160–169.
33. M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 166–181, 2005.
34. D. Hou and L. Mo, "Content categorization of api discussions," in *ICSM*, 2013, pp. 60–69.
35. W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Trans. Software Eng.*, vol. 39, no. 9, pp. 1264–1282, 2013.
36. L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of API documentation," in *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE 2011)*, March-April 2011, pp. 416–431.
37. M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 83–94.
38. M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

39. S. Thummalapenta and T. Xie, "Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 327–336.
40. D. Kawrykow and M. P. Robillard, "Improving api usage through automatic detection of redundant code," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–122.
41. E. Moritz, M. L. Vásquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers, "Export: Detecting and visualizing api usages in large source code repositories," in *ASE*, 2013, pp. 646–651.
42. D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-aware configuration management for object-oriented programs," in *ICSE '07: Proceedings of the 29th international conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2007, pp. 427–436.
43. M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 151–160.
44. T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 70–79.
45. T. Tonelli, K. Czarnecki, and R. Lämmel, "Swing to SWT and back: Patterns for API migration by wrapping," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania.* IEEE Computer Society, 2010, pp. 1–10.
46. 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.
47. T. Espinha, A. Zaidman, and H.-G. Gross, "Web api growing pains: Stories from client developers and their code," in *CSMR-WCRE*, 2014, pp. 84–93.
48. S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: a study of the maven repository," in *14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2014). Victoria (Canada), 28-29 Sept. 2014.* IEEE Computer Society, 2014.
49. Z. Xing and E. Stroulia, "Differencing logical uml models," *Autom. Softw. Eng.*, vol. 14, no. 2, pp. 215–259, 2007.
50. T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *ICSE '08: Proceedings of the 30th international conference on Software engineering.* New York, NY, USA: ACM, May 2008, pp. 471–480.
51. 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.
52. J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting apis with examples: Lessons learned with the apiminer platform," in *WCRE.* IEEE, 2013, pp. 401–408.

Table 16: API change and usage related studies 2-1

		Goal	Target System	API change types	API change type distribution	Detecting Tool	API usage types	API usage type distribution	API change and usage relation	Internal vs. third party usage
API Changes	Raemaekers *et al.* [48]	To investigate if the developers of the frameworks in Maven central repository follow the semantic versioning rule	22,205 projects in Maven central repository	10 types of breaking and 10 types of non-breaking changes	Yes	Clirr	No	No	No	No
	Hou and Yao [3]	To classify API changes according to domains and design intentions.	Swing and AWT APIs	Deprecated, new added, overriden APIs, and subtypes according the the chagne intention	No	Manual analysis	No	No	No	No
	Cossette and Walker [21]	To investigate how the APIs are changed between the releases of frameworks	JDOM, Log4j, and Struts	Fully automatable, partially automatable, and hard to automate according to the easiness to adapt	No	Manual analysis	No	No	No	No
	Dig *et al.* [15]	To present an algorithm detecting refactorings	Eclipse UI, JHotDraw, and Struts	Seven types of refactorings	No	Refactoring Crawler	No	No	No	No
	Xing and Stroulia [23]	To present an approach recommending replacements for changed APIs	HTIMUnit and JFreeChart	30+ API changes causing API migration problems	No	Diff-CatchUp	No	No	No	No

Table 17: API change and usage related studies 2-2

		Goal	Target System	API change types	API change type distribution	Detecting Tool	API usage types	API usage type distribution	API change and usage relation	Internal vs. third party usage
API usages	Lämmel *et al.* [7]	To investigate how framework APIs are used on a large scale	6,286 projects in SourceForge repository	No	No	Self-developed AST-Based tool	Framework-like and library-like API usages	Yes	No	No
	Businge *et al.* [6]	To investigate how third-party Eclipse plug-ins use Eclipse APIs.	512 Eclipse third-party plug-ins in SourceForge repository	No	No	Self-developed tool and Maunal analysis	Official API and internal API usage	Yes	No	No
Both	Robbes *et al.* [9]	To investigate how developers react to API deprecation	2,600 systems of two software ecosystems in Smalltalk	Addition, removal, and modification of classes and methods marked as deprecated	No	Self-developed tool and Maunal analysis	No	No	Yes	No
	Dietrich *et al.* [4]	To investigated the differences between Java compile-time and link-time compatibility and their influences on client programs.	109 programs from the Qualitas corpus [5]	Binary incompatible but source compatible, binary compatible but source incompatibel, and both incompatible chagnes	Yes	ASM and JaCC	No	No	Yes	No
	This study	To investigate API changes and usages together on large scale	36,000 projects in Maven central repository and 16 releases of Eclipse SDK Classic	23 types based on the classification of Des˜Rivières[2]	Yes	ACUA	API-injection usages and others	Yes	Yes	Yes