ON THE USE OF REPLACEMENT MESSAGES IN API DEPRECATION: AN EMPIRICAL STUDY

GLEISON BRITO BATISTA

ON THE USE OF REPLACEMENT MESSAGES IN API DEPRECATION: AN EMPIRICAL STUDY

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Tulio de Oliveira Valente
Co-Advisor: Andre Cavalcante Hora

Belo Horizonte

June 2016

Brito Batista, Gleison

D1234p On the Use of Replacement Messages in API Deprecation: An Empirical Study / Gleison Brito Batista. — Belo Horizonte, 2016
xvi, 57 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Marco Tulio de Oliveira Valente

1. . I. Título.

CDU 519.6*82.10

[Folha de Aprovação]

Quando a secretaria do Curso fornecer esta folha,
ela deve ser digitalizada e armazenada no disco em formato gráfico.

Se você estiver usando o pdflatex,
armazene o arquivo preferencialmente em formato PNG
(o formato JPEG é pior neste caso).

Se você estiver usando o latex (não o pdflatex),
terá que converter o arquivo gráfico para o formato EPS.

Em seguida, acrescente a opção approval={*nome do arquivo*}
ao comando \ppgccufmg.

Se a imagem da folha de aprovação precisar ser ajustada, use:
approval=[*ajuste*][*escala*]{*nome do arquivo*}
onde *ajuste* é uma distância para deslocar a imagem para baixo
e *escala* é um fator de escala para a imagem. Por exemplo:
approval=[-2cm][0.9]{*nome do arquivo*}
desloca a imagem 2cm para cima e a escala em 90%.

Resumo

Como quaisquer sistemas de software, *frameworks* e bibliotecas evoluem ao longo do tempo, assim como suas APIs. Consequentemente, sistemas clientes devem ser constantemente atualizados para utilizarem APIs melhoradas. Para facilitar essa tarefa e preservar a compatibilidade com versões anteriores, elementos de API devem ser depreciados com mensagens de substituição. No entanto, na prática, existem evidências de que esses elementos são usualmente depreciados sem tais mensagens. Nessa dissertação, são estudados um conjunto de questões relacionadas à adoção de mensagens de depreciação. O trabalho objetiva: (i) mensurar a utilização de mensagens de depreciação e (ii) investigar a necessidade de uma ferramenta para recomendar essas mensagens. Para tanto, foram verificados (i) a frequência de elementos depreciados com mensagens de depreciação, (ii) o impacto da evolução de software nessa frequência e (iii) as características dos sistemas com elementos depreciados corretamente. Para alcançar esses objetivos, foi realizado um estudo com 622 sistemas Java e 229 sistemas C#. Esse estudo mostrou que: (i) 66,7% dos elementos de APIs de um sistema são depreciados com mensagens de substituição em Java; para C# esse valor é 77,8%, (ii) em ambas linguagens há pouco esforço para melhorar as mensagens de depreciação ao longo do tempo e (iii) sistemas que depreciam elementos de API corretamente são estatisticamente diferentes em termos de tamanho, comunidade de desenvolvedores e atividade. Também foi realizado um segundo estudo para avaliar a viabilidade de uma ferramenta de recomendação capaz de inferir mensagens de substituição utilizando as soluções adotadas pelos desenvolvedores. Como resultado temos que: (i) 73% das recomendações sugeridas pela ferramenta correspondem de fato a mensagens de substituição reais de elementos de API depreciados e (ii) os percentuais de mensagens de substituição cobertos pela ferramenta em três sistemas relevantes são 28,2%, 30,7% e 37,5%. Os resultados obtidos apontam que essa ferramenta pode oferecer sugestões úteis para mantenedores de software.

Keywords: Depreciação de APIs, Evolução de software, Manutenção de software.

Abstract

As any software system, frameworks and libraries evolve over time, and so their APIs. Consequently, client systems should be updated to benefit from improved APIs. To facilitate this task and preserve backward compatibility, API elements should be deprecated with clear replacement messages. However, in practice, there are evidences that API elements are usually deprecated without such messages. In this dissertation, we study a set of questions regarding the adoption of deprecation messages. Our goal is twofold: to measure the usage of deprecation messages and to investigate whether a tool is needed to recommend such messages. Thus, we verify (i) the frequency of deprecated elements with replacement messages, (ii) the impact of software evolution on such frequency, and (iii) the characteristics of systems with API elements deprecated in a correct way. To achieve these goals we perform an empirical study using 622 Java systems and 229 C# systems. Our large-scale analysis shows that (i) 66.7% of the API elements in Java are deprecated with replacement messages per system, and for C# this value is 77.8%, (ii) there is almost no major effort to improve deprecation messages over time in both languages, and (iii) systems that deprecate API elements in a correct way are statistically significantly different in terms of size, developing community, and activity. In addition, we perform a second study to evaluate the feasibility of a recommendation tool to infer replacement messages by mining solutions adopted by developers. As result of this second study, we report that: (i) 73% of the recommendations provided by the tool correspond to the real replacement messages of deprecated API elements and (ii) the percentage of replacement messages covered by the tool in three relevant systems are 28.2%, 30.7%, and 37.5%. These results suggest that this tool can provide real value to software maintainers.

Keywords: API deprecation, Software evolution, Software maintenance.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This chapter introduces this master dissertation. Initially, we discuss the motivations of the work in Section 1.1. Section 1.2 details the proposed study. Section 1.3 presents our publications. Finally, Section 1.4 presents the outline of this study.

1.1 Motivation

Nowadays, it is a common practice to implement systems on top of frameworks and libraries [Tourwé and Mens, 2003], taking advantage of their *Application Programming Interfaces* (APIs). This practice provides several benefits to client systems, for example:

- Reduction of development costs and time [Moser and Nierstrasz, 1996];
- Increase focus on the essential system requirements, since developers do not need to re-implement the services provided by an API [Konstantopoulos et al., 2009];
- Increase software quality by using well-adopted, tested and documented code elements.

However, as any software system, frameworks/libraries and their APIs evolve over time. Naturally, public types, methods, and fields provided by these APIs may be renamed, removed or updated. Consequently, client systems should migrate to benefit from improved API elements.

To facilitate client developers making the transition and preserve backward compatibility, API elements should be deprecated with replacement messages. Mechanisms to support API deprecation are provided by most programming languages, such Java and C#. For example, Java has two solutions to deprecate types, methods, and fields:

using deprecation annotations and/or deprecation Javadoc tags. Both annotations and Javadoc tags are used to warn developers referencing deprecated API elements. However, the latter may be accompanied by replacement messages to suggest what to use instead. Listing 1.1 presents an example of a deprecated method in Java. In this example, the *getPostParams* method is deprecated with a *@Deprecated* annotation (line 4) and a Javadoc tag *@deprecated* (line 2). This tag contains a replacement suggestion for the deprecated method, which is, in the presented example, the method *getParams*.

```
1 /**
2  * @deprecated Use {@link #getParams()} instead.
3  */
4 @Deprecated
5 protected Map<String, String> getPostParams() throws AuthFailureError {
6     //...
7 }
```

Listing 1.1. Deprecated method in Java - GOOGLE/IOSHED

In a similar way, the standard solution to deprecate API elements in C# consists on using the *Obsolete* attribute accompanied by a replacement message. Listing 1.2 shows an example of a method deprecated with the *Obsolete* attribute (line 1). This attribute includes a parameter with the deprecation message.

```
1 [Obsolete("Use GenerateAsync instead")]
2 public static void Generate(PythonTypeDatabaseCreationRequest request) {
3     //...
4 }
```

Listing 1.2. Deprecated method in C# - MICROSOFT/PTVS

In practice, previous studies indicate that API elements are often deprecated with missing or unclear replacement messages. Robbes et al. [2012] perform a large-scale study in order to investigate the impact of API deprecation in a large-scale software ecosystem. In this study, the authors present preliminary evidences that APIs are usually deprecated without replacement messages. Hora et al. [2015] investigate the impact of API evolution also at an ecosystem level. Their results suggest that deprecation mechanisms should be more adopted. However, we are still unaware about the scale of this phenomenon and whether it tends to get better (or worse) over time.

1.2 Proposed Work

In order to investigate the adoption of API deprecation messages, we analyse in this master dissertation the following factors:

- The frequency of deprecated API elements with replacement messages;

- The impact of software evolution on the frequency of replacement messages;
- The characteristics of systems which deprecate API elements in a correct way in terms of popularity, size, community, activity, and maturity.

Our goal is twofold: to measure the usage of deprecation messages and to investigate whether a tool is needed to recommend such messages. Thus, we propose the following research questions to support our study:

- ***RQ #1. What is the frequency of deprecated APIs with replacement messages?*** In this research question, we analyse the frequency of deprecated API elements with replacement messages, using a large dataset of Java and C# systems. We detected that 66.7% of the API elements are deprecated with replacement messages per system in Java. For C#, this value is 77.8%;
- ***RQ #2. What is the impact of software evolution on the frequency of replacement messages?*** In this research question, we analyse the frequency of replacements messages by comparing (i) two releases in a first analysis and (ii) multiple releases in a second one. Overall, we detect that, in both languages, there is almost no major effort to improve such messages over time;
- ***RQ #3. What are the characteristics of software systems with high and low frequency of replacement messages?*** In this research question, we investigate whether system popularity, size, community, activity, and maturity have an impact on the way developers deprecate API elements. We find that systems that follow best deprecation practices are statistically significant different from the ones that do not in terms of size, developing community, and activity.

These research questions are answered in the context of a large-scale analysis based on 622 Java systems and 229 C# systems. In order to investigate the practical implications of the main study, we also perform a second study to investigate the feasibility of designing and implementing a recommendation tool that automatically infers replacement messages by mining real solutions adopted by developers. To support this second study, we investigate the precision and recall of such tool. In this context, we computed a precision of 73%. That is to say, our practical implication analysis shows that we are able to correctly infer missing replacement messages in almost 3/4 of the cases. In order to evaluate recall, we restricted our analysis to three relevant systems. Detected recalls are: 28.2%, 30.7%, and 37.5%. Therefore, these numbers suggest that a recommendation tool targeting elements deprecated without replacement messages is indeed possible to be implemented, with good precision and reasonable recall.

To conclude, the major contributions of this dissertation are summarized as follows:

- We provide a large-scale empirical study to better understand to what extend APIs are deprecated using replacement messages.
- We provide evidences on the benefits of a recommendation tool to assist client developers in the detection of missing replacement messages.

1.3 Publications

This master dissertation generated the following publications and therefore contains material of them:

- Brito, G., Hora, A., and Valente, M. T. (2015). Um estudo sobre a utilização de mensagens de depreciação de APIs. In *III Workshop de Visualização, Evolução e Manutenção de Software (VEM)* (***best paper***)
- Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016). Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)* (*Qualis A2*)

1.4 Outline of the Dissertation

We organized the remainder of this work as follows:

- **Chapter 2** covers central concepts related to this master dissertation. In the first part, we explain the concept of Application Programming Interfaces. In the second part, we describe how Java and C# systems deprecate API elements.
- **Chapter 3** describes in detail the selection of case studies, the metrics used to investigate API deprecation, the guidelines used to extract replacement messages, and the metrics possibly impacting API deprecation, in order to answer the proposed research questions.
- **Chapter 4** answers and discusses the research questions proposed in this dissertation. First, we discuss the frequency of deprecated APIs with replacement messages. After, we investigate the impact of software evolution on the frequency

of such messages. Finally, we describe the characteristics that impact on the way developers deprecate API elements.

- **Chapter 5** evaluates possible practical implications of the study presented in this master dissertation. First, we explain the motivation of this study and present the study design. Finally, we present preliminary results.
- **Chapter 6** presents final considerations, including related work, a summary of our contributions, and suggestions of future work.

Chapter 2

Background

In this chapter, we provide a general review of the central topics required to understand the work presented in this dissertation. Section 2.1 explains the concept of API and Section 2.2 describes how Java and C# systems deprecate API elements.

2.1 Application Programming Interfaces

In this dissertation, we define Application Programming Interfaces (APIs) as interfaces used by software components to communicate with each other. Figure 2.1 presents a software entity connecting with clients via its API. Examples of successful APIs include Java API[1], .NET Framework Class Library[2], and Android API[3]. More specifically, we define API elements as public/protected types, fields or methods. In addition, C# provides properties, which contain a mechanism to read or write the value of a field. Listing 2.1 shows an example of property. This example shows the declaration of an integer property *MaximumPendingSessions* (line 1), where the methods *get* and *set* are used to return and assign its value, respectively. In order to maintain compatibility between the two languages, we consider properties as fields elements.

```
1 int MaximumPendingSessions
2 {
3     get;
4     set;
5 }
```

Listing 2.1. Example of property in C# - MONO/MONO

[1]https://docs.oracle.com/javase/8/docs/api/
[2]http://msdn.microsoft.com/en-us/library/gg145045.aspx
[3]http://developer.android.com/é-éreference

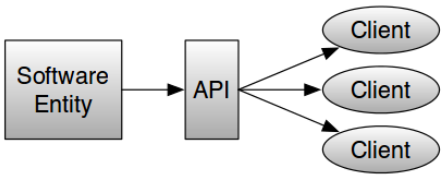

Figure 2.1. APIs acting as interfaces between clients and a provider software entity [Montandon, 2013].

APIs bring many benefits to modern software development [Konstantopoulos et al., 2009; Moser and Nierstrasz, 1996] such as (i) reduction of costs and time, (ii) increase focus on the essential system requirements, since developers do not need to re-implement the services provided by an API, and (iii) increase software quality, since APIs are usually developed by experts and are widely tested.

Listing 2.2 shows an example of API, which presents an excerpt of the *Stack* class in package *java.util* for Java Platform Standard Edition[4]. In this example, the methods *push*, *pop*, *peek*, *empty*, and *search* are *public*, therefore they are defined as API elements. By contrast, *serialVersionID* field (line 18) is *private*, thus it is not an API element. Furthermore, the class *Stack* is also an API element, because it is public.

```
1  public class Stack<E> extends Vector<E> {
2
3      public E push(E item) {
4          //...
5      }
6      public synchronized E pop() {
7          //...
8      }
9      public synchronized E peek() {
10         //...
11     }
12     public boolean empty() {
13         //...
14     }
15     public synchronized int search(Object o) {
16         //...
17     }
18     private static final long serialVersionUID = 1224463164541339165L;
19 }
```

Listing 2.2. Example of API elements in class java.util.Stack from Java

[4]http://www.oracle.com/technetwork/java/javase

2.2 API Deprecation

Software systems evolve over time, changing their elements *e.g.,* methods, fields, and types, and so their APIs. When an API element changes, the impact propagates to client systems. To mitigate the impact of these changes, libraries and frameworks should use deprecation mechanisms, like messages to support developers. In fact, API deprecation is a way to alleviate the impact of API changes.

In theory, before being removed, API elements should be annotated as deprecated to support client developers making the transition to new ones. Deprecated API elements are kept in the system to preserve backward compatibility, but they should not be used by developers because they may be removed in the future.

Many studies investigate the impact of API evolution. For example, through an investigation on the Android ecosystem about adoption of API changes, McDonnell et al. [2013] found that when changes occur rapidly, clients have difficulty in adopting updates. In a large-scale study, Robbes et al. [2012] detected that some deprecation of APIs have large impact on the Smaltalk ecosystem, finding evidence that APIs are usually deprecated with missing and unclear messages. In addition, Hora et al. [2015] investigated the impact of API replacement and improvement on a large-scale ecosystem also written in Smalltalk. The results of this study confirm the large impact on client systems, recommending that deprecation mechanisms should be more adopted.

2.2.1 API Deprecation in Java

The Java language, since J2SE 5.0, provides a mechanism to deprecate types, methods, and fields, using the *@Deprecated* annotation. This annotation causes the compiler to issue a warning when it finds references to deprecated API elements. Listing 2.3 shows an example of a deprecated method using the *@Deprecated* annotation. In this example, the *Database* method is deprecated with this annotation (line 1), but it is not included any suggestion for a replacement.

```
1 @Deprecated
2 public Database(Context context) {
3 //...
4 }
```

Listing 2.3. Deprecated method in Java using *@Deprecated* annotation - FACEBOOK/STETHO

When an element is annotated with the *@Deprecated* annotation, the compiler will issue a deprecation warning if the deprecated element is used (*e.g.,* invoked, referenced, or overridden). The compiler will complain as the message shown in Listing 2.4:

```
1 Note: Path\to\java\file.java uses or overrides a deprecated API.
2 Note: Recompile with -Xlint:deprecation for details.
```

Listing 2.4. Warning message caused by *@Deprecated* annotation

When using deprecation annotations, it is a good practice to document the reasons for the deprecation and/or to recommend alternative API elements. In order do support this practice, Java provides the Javadoc tag *@deprecated* (supported since J2SE 1.1). The tag should also be used to warn developers about deprecated elements. Listing 2.5 shows an example of a deprecated method using the *@deprecated* tag (line 3). The tag contains a deprecation message that suggests replacing the deprecated method *setFieldOrder* (lines 6-8) by the method *getFieldOrder*.

```
1 /**
2 * Force a compile-time error on the old method of field definition
3 * @deprecated Use the required method getFieldOrder() instead to
4 * indicate the order of fields in this structure.
5 */
6 protected final void setFieldOrder(String[] fields) {
7 //...
8 }
```

Listing 2.5. Deprecated method in Java using the Javadoc tag *@deprecated* - JAVA-NATIVE-ACCESS/JNA

More specifically, Java documentation recommends the use of two solutions to deprecate elements with replacement messages, *i.e.,* message to suggest developers what to use instead, as follows:

- *Javadoc 1.1*: This Javadoc version recommends to use the annotation *@see* to indicate the replacement API.
- *Javadoc 1.2 and later*: These versions recommend to use the word *use* followed by the annotation *@link* to indicate the replacement API.

Listing 2.6 presents an example of deprecated Java type using the Javadoc 1.1 guideline. In this example, the tag *@deprecated* (line 2) is used to generate documentation in order to help the developers. The tag *@see* (line 3) reports the replacement element. Furthermore, this tag is used to generate the *See Also* field in Javadoc documentation. We also note the use of *@Deprecated* annotation to show the deprecation warning. The warning message is "since 1.4".

Listing 2.7 shows an example of deprecated Java method, according to Javadoc 1.2 guideline. Like in Javadoc 1.1, the tag *@deprecated* (line 3) is used to provide documentation in order to help the developers, but we also see the use for the *use* keyword to indicate the replacement element. Notice the use of the *@link* tag to provide

a link to the documentation of the replacement element. The warning message "Use @link #ScriptSortBuilder(Script, String) instead" will therefore be presented when developers compile a system that calls the deprecated method *lang*. This message contains the suggestion for the replacement element and the link for its documentation.

```
1 /**
2 * @deprecated since 1.4
3 * @see org.apache.accumulo.core.client.IteratorSetting.Column
4 */
5 @Deprecated
6 public class PerColumnIteratorConfig {
7     //...
8 }
```

Listing 2.6. Deprecated type in Java using Javadoc 1.1 guidelines - SPRING-PROJECTS/SPRING-FRAMEWORK

```
1 /**
2  * The language of the script.
3  * @deprecated Use {@link #ScriptSortBuilder(Script, String)} instead.
4  */
5 @Deprecated
6 public ScriptSortBuilder lang(String lang) {
7     //...
8 }
```

Listing 2.7. Deprecated method in Java using Javadoc 1.2 guidelines - ELASTIC/ELASTICSEARCH

Last but not least, Java deprecation guidelines are not mandatory; developers may adopt other conventions to create replacement messages, or simply do not use them at all.

2.2.2 API Deprecation in C#

The C# language proposes the attribute Obsolete to deprecate elements. Like *@Deprecated* annotations in Java, when an element has the Obsolete attribute, the C# compiler issues a message if it is used. The attribute contains two arguments: (i) a message to support developers and (ii) a parameter to define if the use of deprecated element should cause a compiler error.

The Obsolete attribute can be used with no arguments, but it is recommended to include an explanation of why the item is obsolete and what API element should be used as a replacement.

Listing 2.8 presents an example of deprecated method in C#. The Obsolete attribute is used with the message parameter. The error message is "Use ApiTaskAsync instead.". In this case the value default in compiler error parameter is true.

```
1 [Obsolete("Use ApiTaskAsync instead.")]
2 protected virtual void ApiAsync(HttpMethod httpMethod, string path, object
      parameters, Type resultType, object userState)
3 {
4     //...
5 }
```

Listing 2.8. Deprecated method in C# using Obsolete attribute with a replacement message - FACEBOOK-CSHARP-SDK/FACEBOOK-CSHARP-SDK

Listing 2.9 shows a second example of a deprecated property. The Obsolete attribute is used with a message parameter and a compiler error parameter. The error message is "Unused. Use ItemType property instead". The value of the compiler error parameter is **true**.

```
1 [Obsolete("Unused. Use ItemType property instead.", true)]
2 public string ItemGroupName
3 {
4     get;
5     set;
6 }
```

Listing 2.9. Deprecated property in C# using Obsolete attribute with replacement message and warning parameter - MICROSOFT/MSBUILD

2.3 Final Remarks

This chapter presented essential concepts for understanding the empirical study described in this master dissertation. Initially, we discussed the API concept. In our study, we define API elements as public/protected types, fields or methods from a class. We also consider the properties elements for C# as field elements, in order to maintain the compatibility. We presented the concept of API deprecation, and how Java and C# developers use this mechanism to deprecate API elements.

Chapter 3

Study Design

In this chapter, we describe our experiment design, which is organized as follow. Section 3.1 describes the selection of case studies. Section 3.2 presents the metrics used to investigate API deprecation in this study. Section 3.3 describes the guidelines used to extract replacement messages. Section 3.4 discusses the metrics possibly impacting API deprecation. Section 3.5 describes the extraction of these metrics. Finally, Section 3.6 presents the final remarks.

3.1 Selecting Case Studies

We analyse Java and C# systems hosted on GitHub, a popular social coding platform. We use three criteria to select these systems: number of stars, releases, and deprecated API elements.

1. **Number of stars.** GitHub provides the stargazer button that allows users to show interest on systems. We select systems with 100 or more stars in order to consider real-world and popular systems only.

2. **Number of releases.** We select systems with three or more public releases available on GitHub. We use this criterion to assess API deprecation evolution.

3. **Number of deprecated API elements with replacement messages.** We select systems with at least one public/protected deprecated API element with replacement message. We use this criterion to filter out systems without replacement messages, which are not in the scope of our study.

Based on this filtering criteria, we selected **622** Java systems and **229** C# systems. To better characterize such systems, Figure 3.1 presents the distribu-

tion of the three aforementioned measures. For Java systems, the number of stars in the first quartile, median, and third quartile is 158, 280, and 593. For C#, the number of stars in the first quartile, median, and third quartile is 162, 300, and 719, respectively. The top-3 Java systems with more stars are ELASTIC/ELASTICSEARCH (12.4K stars), NOSTRA13/ANDROID-UNIVERSAL-IMAGE-LOADER (9.7K), and GOOGLE/IOSCHED (7.7K). For C#, the top-3 systems with more stars are DOTNET/COREFX (9.2K), SIGNALR/SIGNALR (5.6K), and DOTNET/ROSLYN (4.5K). For Java, the number of releases in the first quartile, median, and third quartile is 12, 25, and 58, while for C# it is 10, 19, and 42. The top-3 Java systems with more releases are JETBRAINS/KOTLIN (2.9K releases), RSTUDIO/RSTUDIO (2.1K), and FREENET/FRED (1.9K), while for C# they are DNNSOFTWARE/DNN.PLATFORM (3.9K releases), AZURE/AZURE-SDK-FOR-NET (487), and MONO/MONODEVELOP (332). Finally, for Java systems, the number of replacement messages in the first quartile, median, and third quartile is 2, 6, and 20, and for C# it is 2, 7, and 18. The top-3 Java systems with more replacement messages are GROOVY/GROOVY-ECLIPSE (2K deprecated API elements with replacement messages), CYANOGENMOD/ANDROID_FRAMEWORKS_BASE (859), and OPENGAMMA/OG-PLATFORM (815); and for C# they are DNNSOFTWARE/DNN.PLATFORM (588 deprecated API elements with replacement message), UMBRACO/UMBRACO-CMS (500), and MONO/MONO (229).

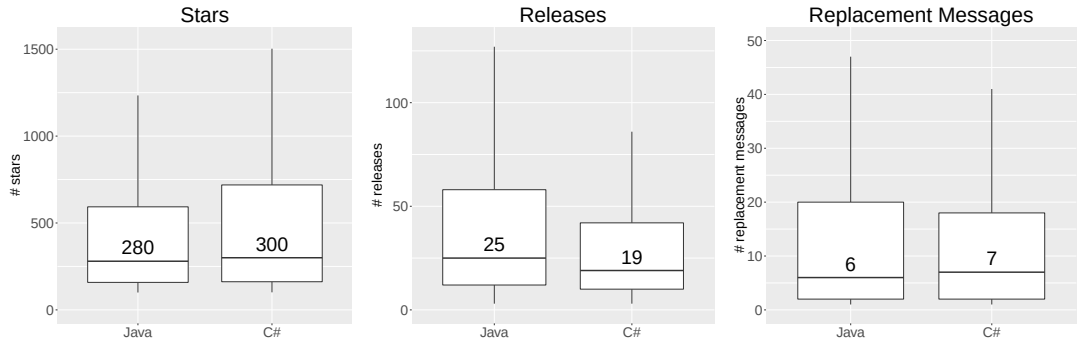

Figure 3.1. Distribution of number of stars, number of releases, and number of deprecated API elements in the selected systems.

In addition, in order to compare the difference between the relative value of messages per system in libraries/frameworks and other systems, we manually classify the selected systems in two categories: *library* and *non-library*. Library systems are defined in this category after inspecting their description on GitHub. We also classify frameworks in this category. Other systems are classified as non-library. We decide to create a separate category for libraries because these systems are supposed to provide more stable, relevant, and used APIs. As noticed in Figure 3.2, for Java, 342 systems

(54.98%) are libraries and 280 systems are non-libraries (45.02%). The values for C# are 119 (51.97%) and 110 (48.03%).

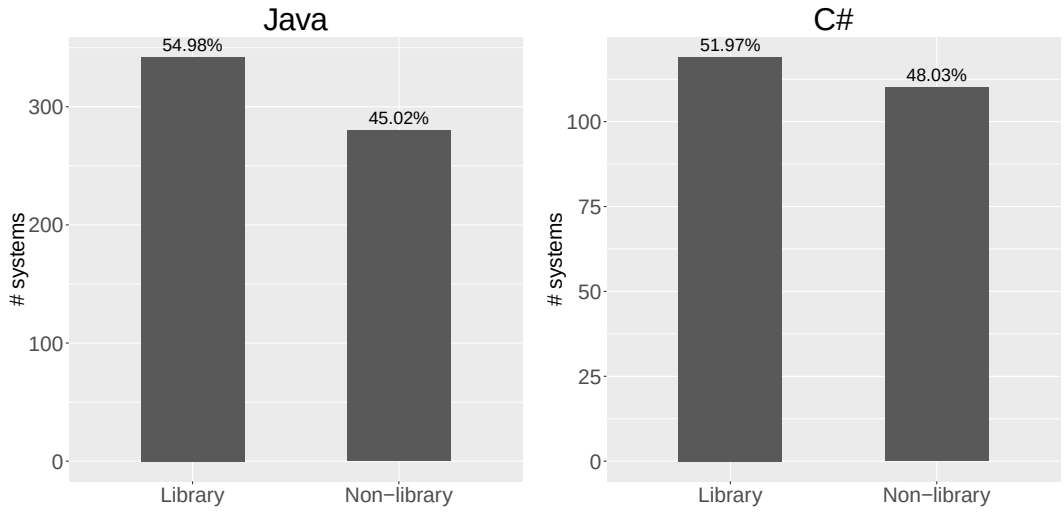

Figure 3.2. Number of systems in library and non-library categories.

3.2 Extracting Deprecated API Elements

As a first step to support answering our research questions, we extract all API elements (including types, fields, and methods) with deprecation annotations in Java and C# systems. For Java, we extract their associated Javadoc and in C# we extract the Obsolete attribute. As presented in Section 2.2, Java API elements are deprecated using the annotation *@Deprecated* and the Javadoc tag *@deprecated*. Listing 3.1 presents an example of a deprecated method Java. In this example, the *@deprecated* annotation (line 6) is used to deprecate a method called *onModule*. Additionally, a Javadoc annotation is used to describe a replacement for the deprecated method instead of calling *onModule*, ELASTIC/ELASTICSEARCH developers should now call *onIndexModule*.

```
1 /**
2 * Old-style guice index level extension point
3 *
4 * @deprecated use #onIndexModule instead
5 */
6 @Deprecated
7 public final void onModule(IndexModule indexModule) {
8 //...
9 }
```

Listing 3.1. Example of obsolete method in Java - ELASTIC/ELASTICSEARCH

C# API elements are deprecated using the Obsolete attribute. Listing 3.2 presents an example of deprecated method in C#. In this example, the *Obsolete* attribute (line 1) is used to deprecate a type called *IHashCodeProvider*. Note that the Obsolete attribute is used to suggest a replacement for the deprecated type. As a substitute of calling *IHashCodeProvider*, MICROSOFT/CODECONTRACTS developers should now call *IEqualityComparer*.

```
1 [Obsolete("Please, use IEqualityComparer insteaded.")]
2 public interface IHashCodeProvider {
3     //...
4 }
```

Listing 3.2. Example of obsolete method in C# - MICROSOFT/CODECONTRACTS

To find deprecated API elements in Java, we implemented a parser based on the Eclipse JDT library to search for deprecation annotations and tags. In order to find deprecated API elements in C#, we implemented an in-house tool based on lexical analysis to detect deprecation attributes. This tool uses regular expressions to identify methods, fields, and types and to check whether they are deprecated. Table 3.1 shows the regular expressions used by the tool. As mentioned in Section 2.1, in our study we consider property elements in C# as fields.

Table 3.1. Regular expressions to identify API elements in C#

Element	Expression
Field	(public|protected)\\s+"(\\w+)=(?:\")? (.*?(?=\"?\\s+\\w+=|(?:\"?)\$))
Property	(\S+(?:<.+?>)?)(?=\s\w+\s\{get;)
Method	((public|protected|static|final|native| synchronized|abstract|transient)+\\s)+ [\\$_\\w\\<\\>\\[\\]]*\\s+[\\$_\\w]+\\ ([^\\)]*\\)?\\s*\\{?[^\\}]*\\}
Type	"\\s*(public|protected)\\s+class\\s+ (\\w+)\\s+((extends\\s+\\w+)| (implements\\s+\\w+(,\\w+)*))?\\s*\\{")

We restricted our analysis to public and protected API elements because they represent the external contracts to clients. Table 3.2 shows the number of public/protected deprecated API elements.

Table 3.2. Number of deprecated API elements.

Language	Deprecated Types	Deprecated Fields	Deprecated Methods	All Deprecated Elements
Java	5,814	4,521	26,727	37,062
C#	1,277	2,197	4,723	8,197

3.3 Extracting Replacement Messages

In Java, when an element is deprecated using the Javadoc tag, it may be accompanied by a replacement message to help client developers. As presented in Section 2.2, Java guidelines propose two solutions to create deprecation replacement messages: (i) using the annotation *@see*, or (ii) using the word *use* and the annotation *@link*. However, to detect alternative guidelines followed by Java developers, we extracted deprecation messages with the support of the JDT library, and we manually inspected a subset of these messages. As a result of this analysis, we detected, in addition to the word *use*, seven frequent words/patterns to indicate replacement: *refer*, *equivalent*, *replace** (*i.e., replace*, *replaced*, *replacement*), *see*, *moved*, *instead*, and *should be used*. We also confirmed that the two frequently adopted annotations are *@link* and *@see*.

Table 3.3 shows the frequency of each replacement guideline found in our manual analysis, as well as message examples. The most adopted guideline is the word *use*, with 17,810 cases (47.9%). In contrast, the least adopted guideline is *should be used*, with 33 cases, (0.09%). Notice that some guidelines may co-occur in the same message. For example, *use* commonly happens with *@link*. In total, 22,032 (59.4%) API elements were deprecated with replacement messages out of the 37,062 elements considered in this study.

For C#, when an element is deprecated with the *Obsolete* attribute, like in Java, it may be accompanied by a replacement message to help client developers. In contrast to Java, we could not find guidelines to define replacement messages for C#. For this reason, we also performed a manual analysis to detect these guidelines in C#. First, we extacted deprecation messages in deprecated API elements. We looked for occurrences of the attribute *Obsolete* and we manually inspected a subset of these messages. As a result, we detected three frequent words/patterns that are used to indicate replacement: *use*, *replace** (*i.e., replace*, *replaced*, *replacement*), and *instead*.

Table 3.4 presents the frequency of each identified replacement guideline in C#, with real examples of replacement messages. The most common guideline is the word *instead*, with 3,118 cases (51%). In contrast, the least adopted guideline is *replace**, with 422 cases (8%). In total, 5,268 (64.3%) API elements are deprecated with

Table 3.3. Frequency of replacement message guidelines in Java.

Guideline	Frequency	Replacement Message Example
use	17,810 (47.9%)	use encodeURL(String url) instead (Apache Tomcat)
replace*	2,171 (5.8%)	Replace to getParameter(String, int) (Dubbo)
refer	1,070 (2.9%)	property will be removed, refer @link #getEncoded(boolean) (Actor Platform)
equivalent	166 (0.3%)	The @link Iterable equivalent is @link ImmutableSet#of() (Google Guava)
see	777 (2.1%)	See servlet 3.0 apis like HttpServletRequest.getParts() (Eclipse Jetty)
moved	224 (0.6%)	deprecated since 2008-05-28. Moved to stapler (Eclipse Hudson)
instead	14,173 (38.2%)	Use KEY_LMETA instead (Facebook Nifty)
should be used	33 (0.09%)	org.bukkit.entity.minecart.PoweredMinecart should be used instead (Bukkit)
@link	14,852 (40%)	Use @link #setController(DraweeController) instead (Facebook Fresco)
@see	2,334 (6.3%)	@see #getStartRequests (WebMagic)

Table 3.4. Frequency of replacement message guidelines in C#.

Guideline	Frequency	Replacement Message Example
use	2,686 (49%)	Use static Add() method instead. (Mono Monomac)
replace*	422 (8%)	Replace it with both GetSupportedInterfaceOrientations (Redth/ZXing.Net.Mobile)
instead	3,118 (51%)	This class is obsolete; use class Tree instead (Mono)

replacement messages. This data is further explored in Research Question #1, and its evolution is analysed in Research Question #2.

3.4 Metrics Possibly Impacting API Deprecation

To support answering Research Question #3, about the characteristics of systems that deprecate API elements with replacement messages, we consider metrics in five dimensions which are likely to affect deprecation practices: popularity, size, community, activity, and maturity. The goal is to investigate whether such metrics have an impact on the way developers deprecate API elements. These metrics are described next and summarized in Table 3.5.

- **Popularity.** This dimension includes metrics that represent how popular is a system in GitHub in *number of stars*, *number of watchers*, and *number of forks*.

The rationale is that popular systems may have more clients, thus their developers might have more concerns about their APIs.

- **Size.** This dimension includes metrics related to system size in terms of *number of files* and *number of API elements* (*i.e.,* sum of number of types, fields, and methods). The rationale is that larger systems are harder to maintain, therefore it might be more difficult to keep track of all API changes. In contrast, smaller systems may be easier to control and to keep track of.

- **Community.** This dimension includes metrics that represent the system community, including *number of contributors*, *average files per contributor*, and *average API elements per contributor*. The rationale is that systems with larger communities might be somehow easier to maintain, and to keep track of API changes.

Table 3.5. Metrics likely to impact API deprecation.

Dimension	Metric
Popularity	number of stars
	number of watchers
	number of forks
Size	number of files
	number of API elements
Community	number of contributors
	average files per contributor
	average API elements per contributor
Activity	number of commits
	number of releases
	average days per release
Maturity	age (in number of days)

- **Activity.** This dimension includes metrics related to the system activity level in terms of *number of commits*, *number of releases*, and *average days per release*. The rationale is that systems with more activity might respond faster to client complains. Therefore, they may be more likely to improve their APIs.

- **Maturity.** This dimension is about the system age, in *number of days*. The rationale is that older systems are reliable, thus they may have stable APIs. In contrast, it is natural to expect that newer systems have less stable APIs.

3.5 Extracting Metrics from Case Studies

We extracted the proposed metrics from two groups of systems: the ones deprecating API elements in a correct way, *i.e.,* by providing replacement messages to deprecated API elements, and the ones not following this practice. Then, we assessed such groups to verify whether they are statistically different with respect to the proposed metrics. These two steps are detailed next.

Selecting systems and extracting metrics. We sorted all systems, in descending order, based on the percentage of deprecated API elements with replacement messages. We selected two groups, top-30% (*i.e.,* systems with the highest percentage of deprecated API elements with replacement messages) and bottom-30% (*i.e.,* systems with the lowest percentage). For Java, each group has 187 systems and for C# each group has 69 systems. Figure 3.3 shows the relative distribution of deprecated elements with replacement messages in each group. In the Java systems, the median percentage is 100% for the *top* systems and 17.8% for the *bottom* systems. For C# systems, the median percentage is also 100% for the *top* systems and 42.5% for the *bottom* systems. Finally, we extracted the metrics described in Section 3.4 for the *top* and *bottom* systems.

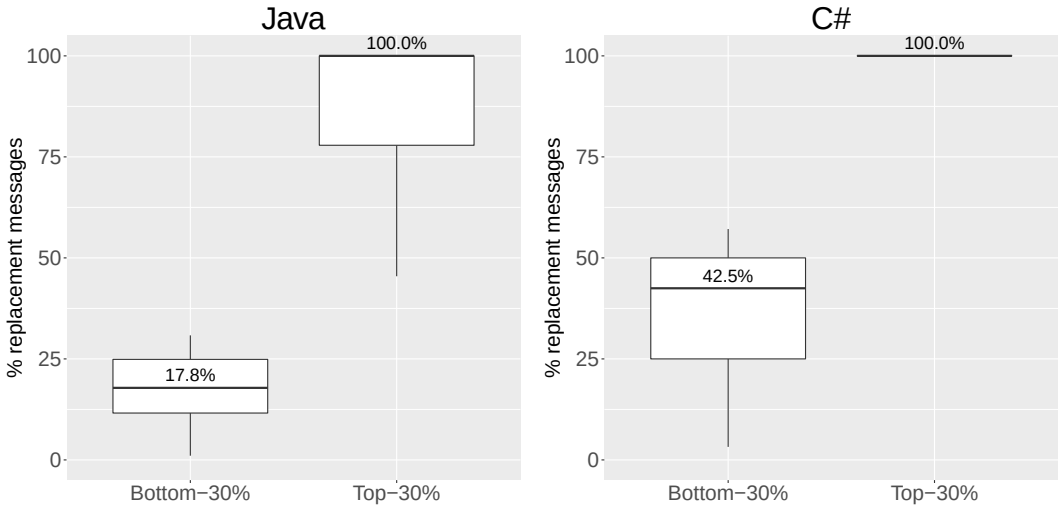

Figure 3.3. Distribution of the percentage of deprecated APIs with replacement messages in the top-30% and bottom-30% systems.

Assessing selected systems. We compare the values of each metric in *top* and *bottom* systems. We first analyse the statistical significance of the difference between the two groups by applying the Mann-Whitney U test at *p-value* = 0.05. To show

the effect size of the difference between the two groups, we compute Cliff's Delta (or d). As in previous studies [Grissom and Kim, 2005; Tian et al., 2015; Linares-Vásquez et al., 2013], we interpret the effect size values as small for $0.147 < d < 0.33$, medium for $0.33 < d < 0.474$, and large for $d > 0.474$.

3.6 Final Remarks

This chapter presented the design of the experiment reported in this master dissertation. First, we describe the selection of case studies. We use three criteria to select the systems: systems with 100 or more stars in GitHub, three or more public releases, and at least one public/protected deprecated API element with replacement messages. At the end of the selection, we retrieved 622 Java systems and 229 C# systems. We also classify the systems in two categories: library and non-library. For Java, the number of library systems is 343 (54.98%) and the number of non-library systems is 280 (45.02%). For C#, the values are 119 (51.97%) and 110 (48.03%), respectively. Next, we presented the guidelines we followed to identify replacement messages. For Java, the guidelines include the use of the expressions *use*, *replace**, *refer*, *equivalent*, *see*, *moved*, *instead*, *should be used*, and the annotations *@link* and *@see*. The guidelines for C# include the expressions *use*, *replace**, and *instead*. We also present metrics we will use to investigate the frequency that developers deprecate API elements. These metrics belong to five dimensions: popularity, size, community, activity, and maturity. Finally, we proposed a division of the systems in two groups, top-30% and bottom-30%, in order to contrast the values of the defined metrics.

Chapter 4

Results

In this chapter, we answer and discuss the three research questions proposed in this study. Section 4.1 discusses the frequency of deprecated APIs with replacement messages. Section 4.2 investigates the impact of software evolution on the frequency of such messages. Section 4.3 describes the characteristics that impact on the way developers deprecate API elements. Section 4.4 presents possible threats to the validity of our study. Finally, in Section 4.5 we present some final remarks, as a conclusion to this chapter.

4.1 RQ1. What is the frequency of deprecated APIs with replacement messages?

In this first research question, we analyze the frequency of deprecated API elements with replacement messages in the Java systems in the last release of the cases studies. As presented in Table 4.1, 3,789 deprecated types (65%) contain replacement messages. For deprecated fields and methods, these numbers are 2,675 (59%) and 15,568 (58.2%), respectively. Considering all deprecated API elements in Java, 22,032 (59.4%) contain replacement messages. We also present the frequency of deprecated API elements with replacement messages in the C# systems. As noticed in Table 4.1, 613 deprecated types (48%) in the C# systems contain replacement messages. For deprecated fields and methods, these numbers are 1,401 fields (63.8%), and 3,254 methods (68.9%), respectively. Considering all deprecated API elements in C#, 5,268 (64.2%) contain replacement messages. Therefore, the results measured for Java and C# are very similar, although C# systems have a slight tendency to include more replacement messages, in relative terms.

Table 4.1. Number of deprecated API elements with replacement messages.

Language	Types	Fields	Methods	All
Java	3,789 (65%)	2,675 (59%)	15,568 (58.2%)	22,032 (59.4%)
C#	613 (48%)	1,401 (63.8%)	3,254 (68.9%)	5,268 (64.2%)

Next, we present the absolute and relative analysis per system. For both Java and C# systems, we consider only the ones that have at least one deprecated type, field, and method.

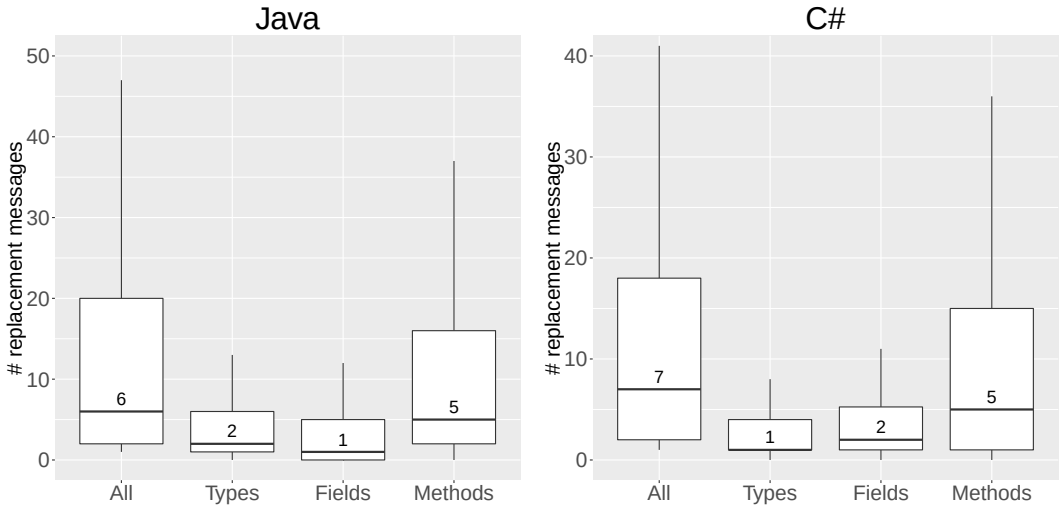

Figure 4.1. Absolute distribution of deprecated API elements with replacement messages.

Absolute analysis. Figure 4.1 shows the distribution of the number of deprecated API elements with replacement messages per system for Java and C#.

For types, in Java, the first quartile is 1, the median is 2, and the third quartile is 6, while in C# we have 1, 1, and 4. Regarding fields elements, for Java the first quartile, median and third quartile are 0, 1, and 5. For C#, these numbers are: 1, 2, and 5. For methods, in Java, first quartile, median, and third quartile are 2, 5, and 16. For C#, these values are 1, 5, and 15, respectively.

Therefore, methods are the most frequently deprecated elements with replacement messages, both in Java and in C#, fields are the least ones in Java (median 1 per system), while types are the least one in C# (median 1 per system). Considering all API elements in Java, the first quartile is 2, the median is 6, and the third quartile is 20. For C#, the values are 2, 7, and 18, respectively.

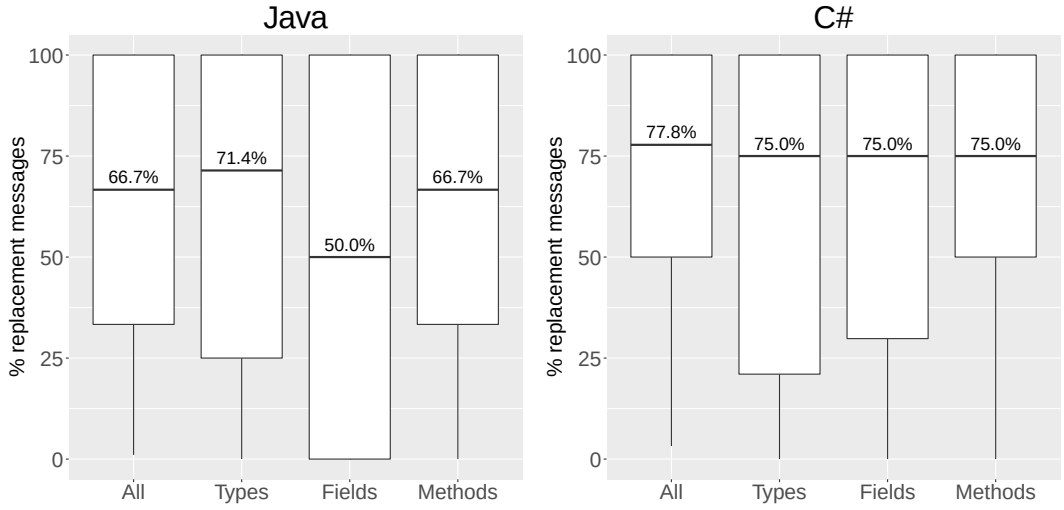

Figure 4.2. Relative distribution of deprecated API elements with replacement messages.

Relative analysis. Figure 4.2 presents the distribution of the relative number of deprecated API elements with replacement messages per system.

In Java, for types, the first quartile is 25%, the median is 71.4%, and the third quartile is 100%. There are 114 systems with 100% of their deprecated types with replacement messages, such as spring-projects/spring-android, jenkinsci/github-plugin, and google/guava. In contrast, we found 57 systems with types deprecated without replacement messages, *e.g.,* caelum/vraptor, cymcsg/UltimateAndroid, and spring-projects/spring-roo. For C#, the first quartile is 21%, the median is 75%, and the third quartile is 100%. We found 61 systems with all deprecated types with replacement messages, such as apache/log4net, elastic/elasticsearch-net, and dotnet/corefx, while there are 30 systems with types deprecated without replacement messages, like mono/monodevelop, ws/aws-sdk-net, and Microsoft/PTVS.

For fields, in Java, the first quartile is 0%, the median is 50%, and the third quartile is 100%. We identify 74 Java systems with 100% of their deprecated fields with replacement messages, like spring-projects/spring-framework, hibernate/hibernate-orm, and apache/tomcat70. In opposition, we found 72 systems without replacement messages in fields, such as goldmansachs/gs-collections, phonegap/phonegap-app-developer, and spring-projects/spring-webflow. For C#, the first quartile, median, and third quartile are 29.8%, 75%, and 100%, respectively. There are 26 systems with 100% of fields with replacement messages, *e.g.,* Azure/azure-storage-net, ServiceS-

TACK/SERVICESTACK, and ASP-NET-MVC/ASPNETWEBSTACK. In contrast, we detect 14 systems with fields deprecated without replacement messages, such as NHIBERNATE/NHIBERNATE-CORE, MICROSOFT/MSBUILD, and MONO/MONO-TOOLS.

For methods, in Java, the first quartile is 33.3%, the median is 66.7%, and the third quartile is 100%. There are 160 Java systems with 100% of their deprecated methods with replacement messages, such as SQUARE/PICASSO, NOSTRA13/ANDROID-UNIVERSAL-IMAGE-LOADER, and ECLIPSE/JGIT. We also found 24 Java systems with no replacement messages, such as JPMORESMAU/ECLIPSEFP, SPRING-PROJECTS/SPRING-DATA-NEO4J, and HIBERNATE/HIBERNATE-VALIDATOR. For C#, the values of the first quartile, median, and third quartile are 50%, 75%, and 100%. We found 63 systems with 100% of their deprecated methods with replacement messages, like GOOGLESAMPLES/CARDBOARD-UNITY, LIBGIT2/LIBGIT2SHARP, and SIGNALR/SIGNALR. There are 20 systems in which methods are deprecated without replacement messages, such as UNITYCONTAINER/UNITY, JOHNNYCRAZY/SPOTIFYAPI-NET, and MICROSOFT/MSBUILD.

For Java, according to the median, types are the most deprecated elements with replacement messages (median of 71.4%), followed by methods (66.7%) and fields (50%). The third quartile at 100% for types, fields, and methods shows that 25% of the systems always deprecate all elements with replacement messages. In contrast, the first quartile at 0% for fields shows that 25% of the systems never deprecate fields with replacement messages. For C#, like in Java, the third quartile is 100% for all elements, and coincidentally, all three elements have the same median values (75%).

When considering all API elements, the first quartile is 33.3%, the median is 66.7%, and the third quartile is 100%, for the Java sytems. In other words, considering the median, around 2/3 of the API elements are deprecated with replacement messages. In Java, there are 162 systems (26%) with 100% of their deprecated API elements with replacement messages, such as CODE4CRAFT/WEBMAGIC, GOOGLE/GUICE, and BUMPTECH/GLIDE. For C#, the first quartile is 50%, the median is 77.8%, and the third quartile is 100%. This means that, considering the median, around 3/4 of the API elements are deprecated with replacement messages. We found 64 systems (28%) with 100% of deprecated API elements with replacement messages, like GOOGLE/GOOGLE-API-DOTNET-CLIENT, APACHE/LOG4NET, and AZURE/AZURE-SDK-FOR-NET. In summary, C# systems, in relative terms, have a slight trend to include more replacement messages, when compared to the Java systems in our dataset.

As described in Subsection 3.1, we classify the systems in two categories: library and non-library. For Java, we found 342 library systems and 280 non-library systems. These numbers for C# are 119 and 110 systems, respectively. In order to compare

the categories, we analyse the relative number of deprecated messages per system in each domain. Figure 4.3 presents the distribution for the categories. For Java, the first quartile, median, and third quartile are 40%, 71.2%, and 100%, for libraries. For non-libraries, these values are 28.6%, 60%, 86.8%, respectively. For C#, the numbers for libraries are 51.3%, 81.8%, and 97.2%. For non-libraries, these values are 50%, 73.7%, and 100%. The difference between the median values of the two categories is 11.2% (Java) and 8.1% (C#). In both languages, the percentage of replacement messages for libraries is greater than for non-libraries (median-values). There is more effort and concern to provide replacement messages in library systems than non-library systems. We also analysed the statistical significance of the difference between the two groups of systems, by applying the Mann Whitney Test at *p-value* = 0.05. For Java systems, the *p-value* is < **0.001**, while in C# the value is **0.596**. In C#, we do not observe a statistical difference between library and non-library categories.

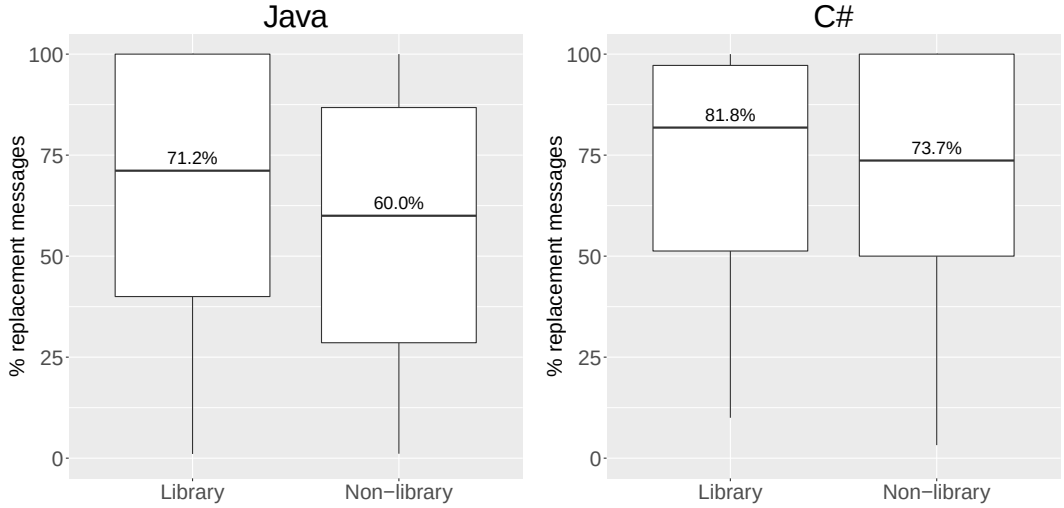

Figure 4.3. Relative distribution of deprecated API elements with replacement messages in library and non-library systems.

Summary: For Java systems, 66.7% of the API elements are deprecated with replacement messages (median measures). This percentage is 71.4% for types, 50% for fields, and 66.6% for methods, suggesting that developers are usually more concerned with types and less with fields. For Java and C#, at least 25% of the systems deprecate types, fields and methods with replacement messages in all elements. We also observe that the number of replacement messages in library systems is greater than in non-library for both languages. Overall, we could not find major differences when comparing the results provided by the Java and C# analysis.

4.2 RQ2. What is the impact of software evolution on the frequency of replacement messages?

In order to study the impact of software evolution on deprecation, we analyze the frequency of deprecated API elements with replacement messages in two distinct releases of the Java and C# systems. We compare the first publicly available release with the last one (*i.e.,* the same releases considered in the Research Question #1). In Java, the number of deprecated APIs elements increases from 10,798 (first release) to 22,032 (last release). In C#, these values are 2,341 and 5,268 API elements, respectively. In addition, in the end of the section, we present a more detailed comparison by taking into account several releases.

Absolute analysis. Figure 4.4 shows the distribution of the absolute number of deprecated API elements with replacement messages per system for Java and C#, in the analysed releases (first and last). For Java, in the first release, the first quartile is 1, the median is 3, and the third quartile is 17. In the last release, the first quartile is 2, the median is 6, and the third quartile is 20. The values for C# are 0, 2, and 9 (first release). For the last release, we have 2, 7, and 18. Therefore, in both languages, there is an increase on the values of first quartile, median, and third quartile measures, from the first to the last considered releases. This is expected due to the natural evolution of the systems, which tend to provide more features and API elements.

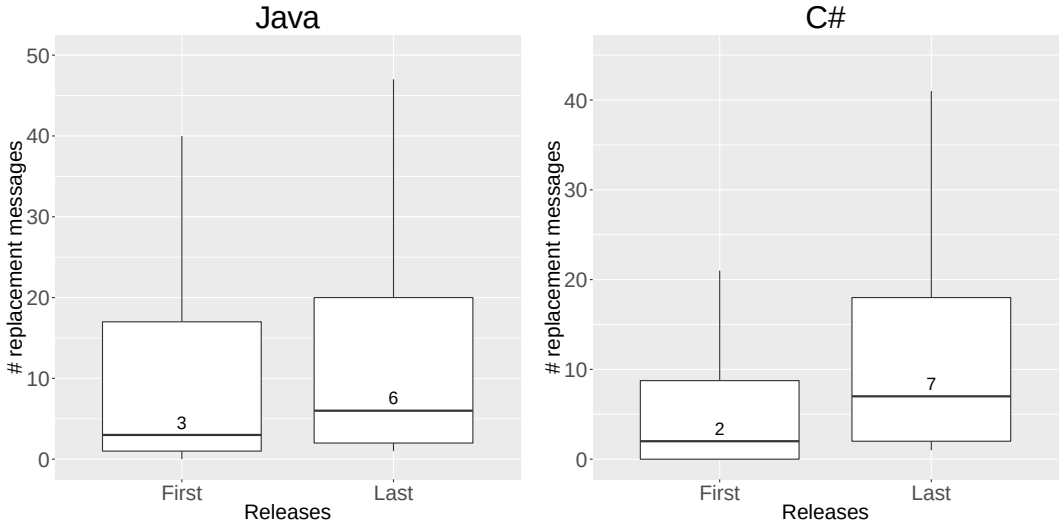

Figure 4.4. Absolute distribution of deprecated API elements with replacement messages over time.

Relative analysis. Figure 4.5 presents the distribution of the relative number of

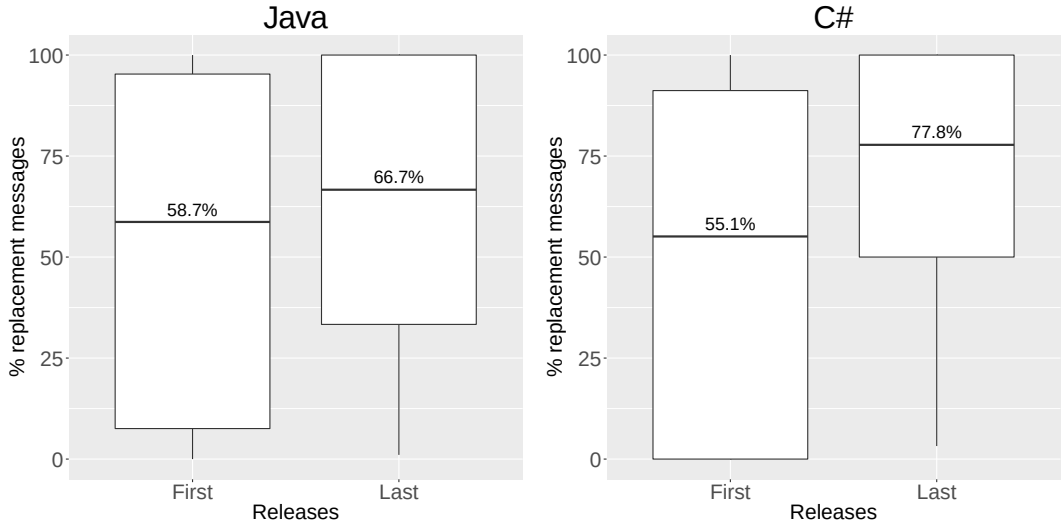

Figure 4.5. Relative distribution of deprecated APIs with replacement messages over time.

deprecated API elements with replacement messages per system, in the two analysed releases. For Java, in the first release, the first quartile is 7.5%, the median is 58.7%, and the third quartile is 95.3%. In the last release, these measures are 33.3%, 66.7%, and 100%, respectively. Therefore, the median increases only by 8% between the considered releases (from 58.7% to 66.7%). We can then infer that there is no major effort from developers to provide deprecation messages. The third quartile also remains almost stable, increasing only 4.7% (from 95.3% to 100%). In contrast, by analysing the evolution of the first quartiles, we observe significant changes. This quartile increases by 25.8% (from 7.5% to 33.3%), meaning that the bottom 25% of the systems are increasingly adopting replacement messages.

For the C# systems, the values of first quartile, median, and third quartile are 0%, 55.1%, and 91.2%. In the last release, we have 50%, 77.8%, and 100%. Therefore, the first release, 25% of the systems do not use replacement messages. This value increases to 50% in the last release. In other words, 25% of the systems started to use replacement messages. The median of the relative number of deprecated API elements with replacement messages increases 22.7% (from 55.1% to 77.8%). Differently from Java, this means that there was a major effort by developers to provide deprecation messages. Similarly to Java, the third quartile keeps at very high percentage, increasing 8.8% (from 91.2% to 100%).

Overall, in Java, from the first to the last release, 121 systems (19.4%) increased the relative number of deprecated API elements with replacement messages. For C# this value is 46 systems (20.1%). Figure 4.6 shows the distribution of the relative

number of deprecated API elements with replacement messages in such systems. The median increased from 19% to 64.3%. Examples of systems in this category include SPRING-PROJECTS/SPRING-BOOT (increased from 0% to 55%) and NETFLIX/EUREKA (increased from 0% to 64%). For C#, the median increases from 40.0% to 81.8%. Examples of such systems include ELASTIC/ELASTICSEARCH-NET (increased from 50% to 91.7%) and HENON/GITSHARP (increased from 0% to 66.7%).

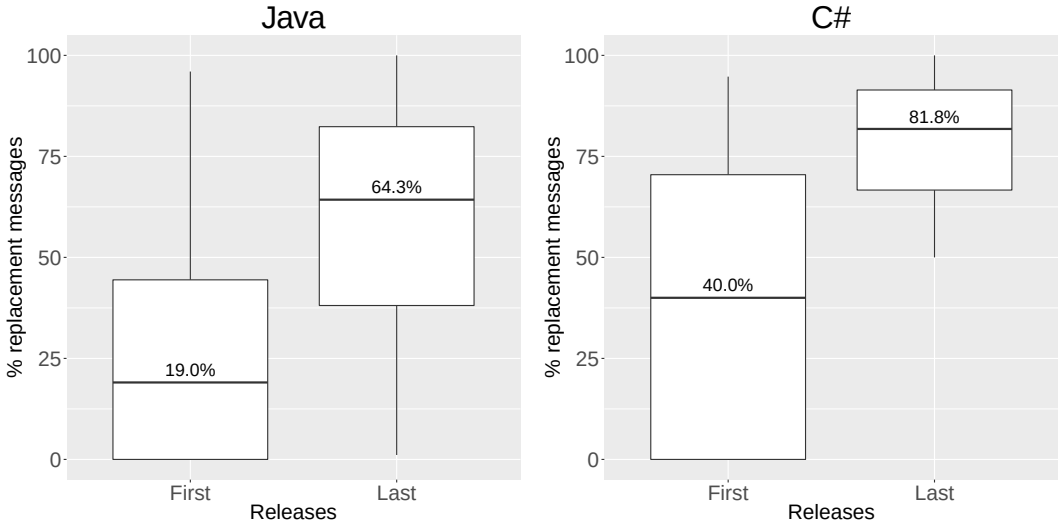

Figure 4.6. Relative distribution of systems where the percentage of replacement messages increased from the first to the last release.

In contrast, in 87 Java systems (14%) the number of deprecated API elements with replacement messages decreased, while for C# this value is 23 systems (10%). Figure 4.7 presents the distribution for such systems. In the Java systems, the median decreases from 84.1% to 53.3%. Examples of systems in this category include APTANA/PYDEV (decreased from 91.6% to 50%) and SPRING-PROJECTS/SPRING-FRAMEWORK (decreased from 98% to 68%). For C#, the median decreases from 91.3% to 69.1%. Examples of systems facing a reduction in the number of replacement messages include DOTNET/COREFX (decreased from 82.7% to 52.5%) and AZURE/AZURE-STORAGE-NET (decreased from 97.1% to 69%).

Finally, in 414 Java systems (66.5%) the number of deprecated API elements with replacement messages remains stable, *i.e.,* it is the same ratio from the first to the last release. Examples include SPRING-PROJECTS/SPRING-BATCH and MCXIAOKE/ANDROID-VOLLEY, both with 100%. For C#, we find 160 stable systems (69.9%). Examples include MONGODB/MONGO-CSHARP-DRIVER and MONO/-MONOGAME, again with 100%.

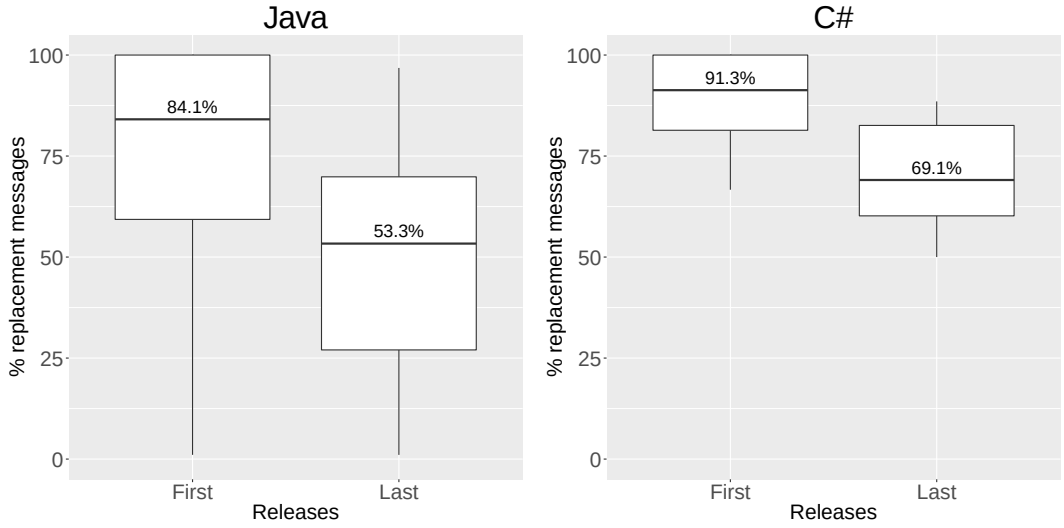

Figure 4.7. Relative distribution of systems where the percentage of replacement messages decreased from the first to the last release.

Analysis using multiple versions. In addition to the previous analysis, where we reported a comparison between two single releases, we provide a second analysis by considering several releases in order to better understand the impact of software evolution on the quality of deprecation messages. Starting from the first release, we collected all the subsequent ones, considering an interval of at least two months. We then classify the variation in the percentage of replacement messages of a given system releases in four categories: (i) **Decrease Trend**: the percentage always decreases in the analysed releases, (ii) **Increase Trend**: the percentage always increases, (iii) **Variant Trend**: the percentage increases and decreases with no pattern, and (iv) **Stable Trend**: the percentage of replacement messages is constant in all releases. Figure 4.8 provides examples of systems in each of the categories.

Table 4.2 presents the number of systems in each category. For Java, 40 systems (6%) are in the decrease category, 198 systems (32%) in the increase category, 245 systems (40%) in variant category, and 139 systems (22%) are stable. The values for C# are 25 systems (11%) for decrease category, 59 systems (26%) for increase, 83 systems (36%) for variant, and 62 systems (27%) for stable. Therefore, most systems are variant, while a small amount has a decrease trend in the percentage of replacement messages. In both languages, only a minority of the systems lose the quality of their deprecation messages over time (*i.e.,* present a decrease trend in the analysed releases).

Figure 4.9 shows the distribution of the number of releases per category. We performed such analysis to reveal the number of releases in the proposed categories of evolution, *i.e.,*, to reveal how the number of releases impacts in the evolution of

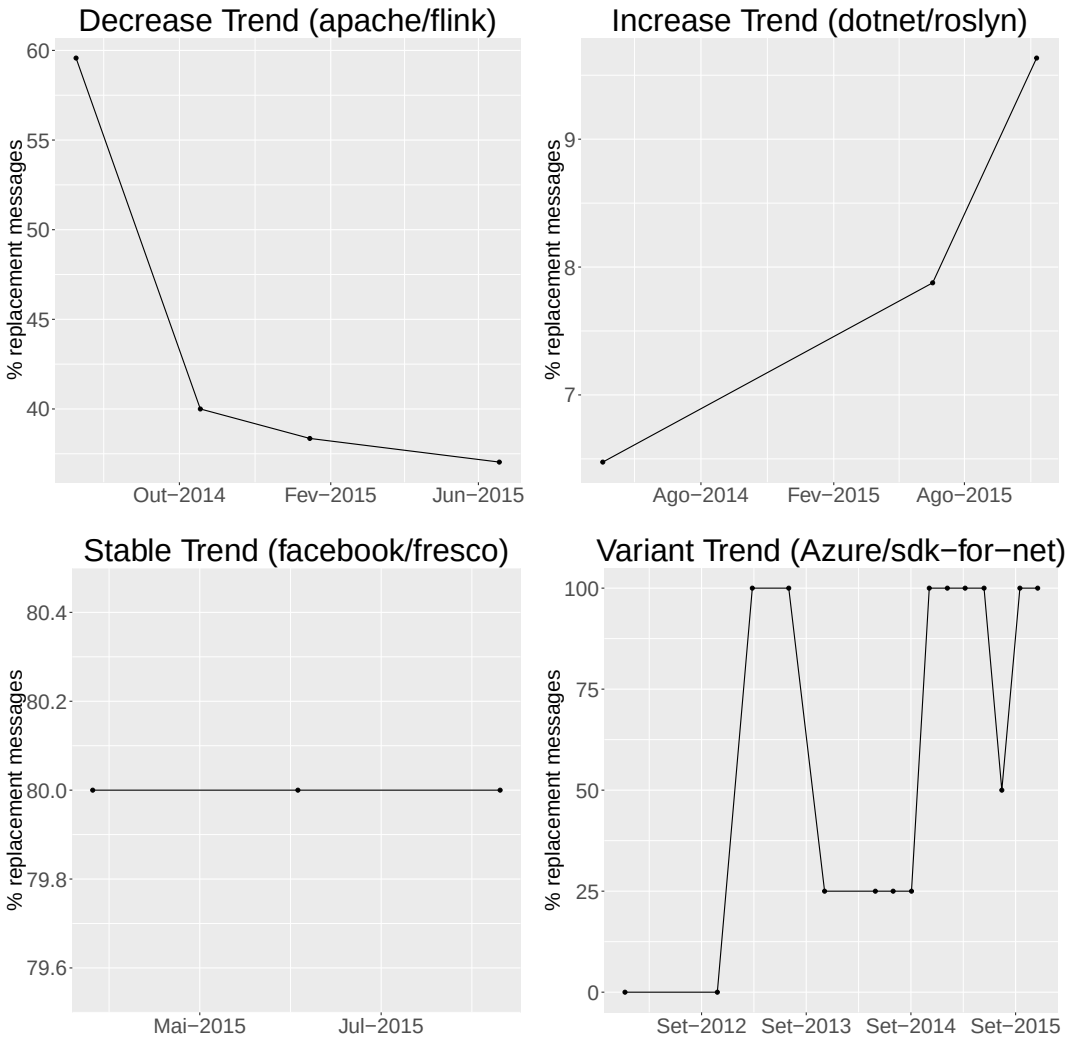

Figure 4.8. Examples of systems in each evolution category.

Table 4.2. Number of systems in each evolution category

Language	Decrease	Increase	Variant	Stable
Java	40 (6%)	198 (32%)	245 (40%)	139 (22%)
C#	25 (11%)	59 (26%)	83 (36%)	62 (27%)

replacement messages. For Java, the values is 18 (decrease trend), 18 (increase trend), 48 (variant trend), and 14 (stable trend). The values in C# is 22 (decrease category), 14 (increase category), 40 (variant category), and 11 (stable category). In both languages, stable systems have the lowest number of releases on the median, while variant systems have the highest value. In fact, systems with more releases are more likely to change their API elements, and consequently, they may show a variation in the percentage of

their deprecation messages over time.

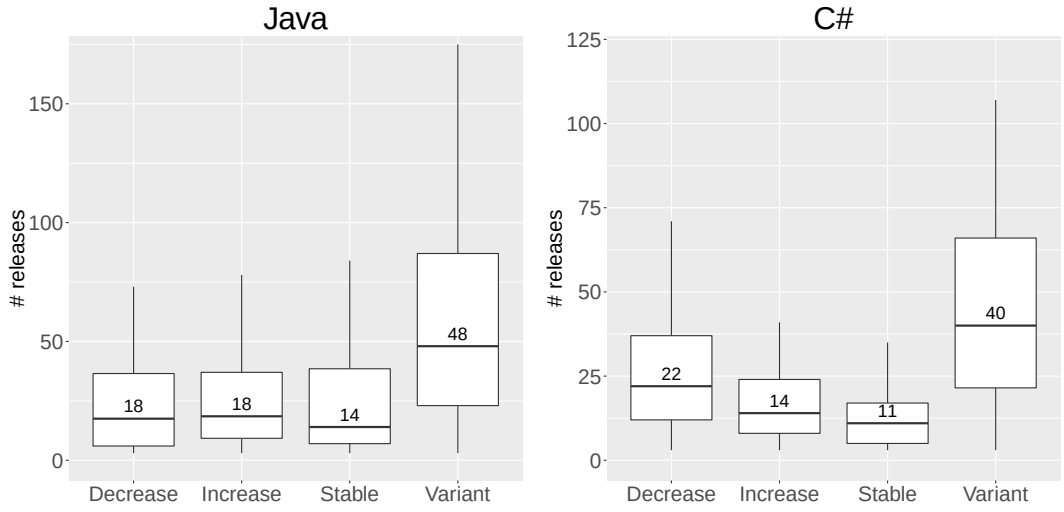

Figure 4.9. Number of releases for each category of system.

Figure 4.10 shows the distribution of the relative number of replacement messages for systems with an increase trend in the percentage of replacement messages, comparing the first and last releases. The value of the first quartile, median, and third quartile for the first release both in Java and in C# is 0%. For the last release in Java, the values are 42.4%, 80%, and 100%. For the last release in C#, the values are 50%, 66.9%, and 97.2%. Considering just the median, Java systems increase 80% (from 0% to 80%), and C# systems increase 67.1% (from 0% to 66.9%). Examples of Java systems in the increase category include Netflix/netflix-graph (increased from 0% to 100%) and paypal/PayPal-Java-SDK (increased from 0% to 60%). For C#, examples include JohnnyCrazy/SpotifyAPI-NET (increased from 0% to 50%) and mongodb/mongo-azure (increased from 0% to 100%). In summary, we note that in the initial release there are no replacement messages at all in such systems. Over time, these systems increase these numbers, meaning their the developers make the decision to provide replacement messages.

Figure 4.11 presents the distribution for systems with a decrease trend, comparing the first and last releases. In Java, the first quartile is 61.3% for the first release, compared to 32.7% for the last one. The median is 93.8% against 50%, and the third quartile is 100% against 68.3%. Examples of systems in this category include liaohuqiu/cube-sdk (decrease from 66.7% to 0%) and evernote/evernote-sdk-android (decrease from 100% to 0%). For C#, the first quartile is 43.3% and 25%, for the first and last releases. The median is 64.7% against 36.4%, and the third quartile is 85.7% against 58.6%. Examples of such systems include dotnet/corefx (decrease

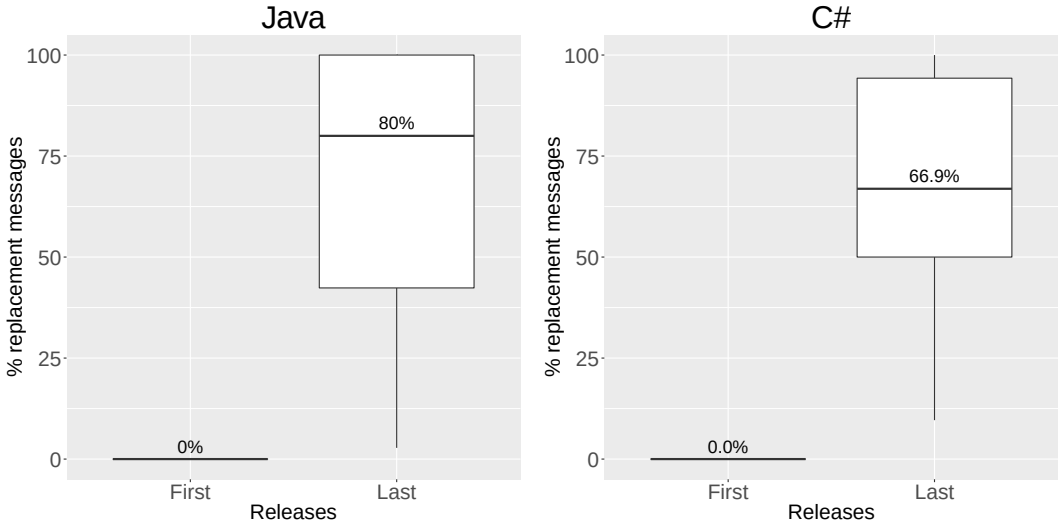

Figure 4.10. Relative distribution of replacement messages in systems in the increase category, comparing the first and last releases.

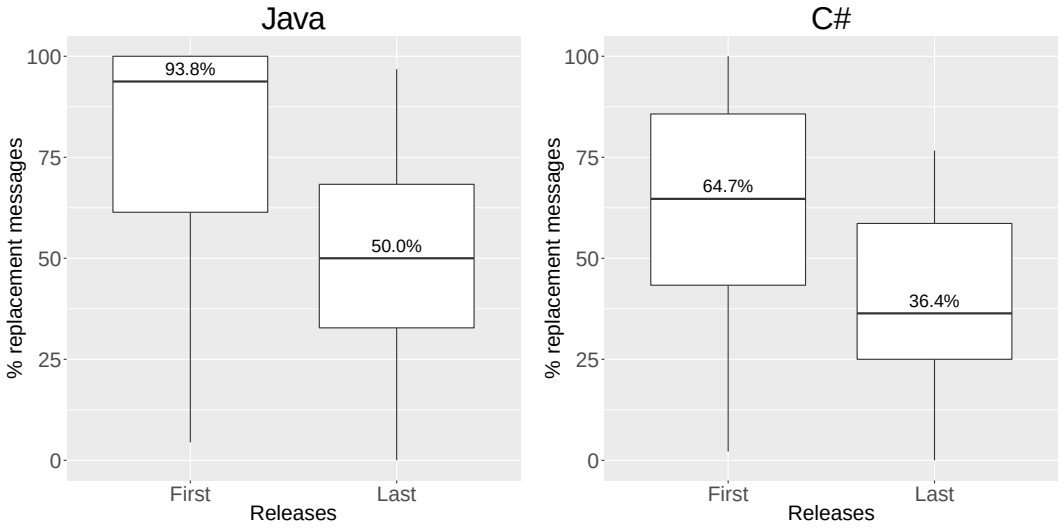

Figure 4.11. Relative distribution of replacement messages in systems in the decrease category, comparing the first and last releases.

from 29.2% to 17.5%) and GOOGLESAMPLES/TANGO-EXAMPLES-UNITY (decrease from 100% to 25%).

Summary: For Java, the relative number of deprecated API elements with replacement messages remain almost constant over time (from 58.7% to 66.7%), showing that there is no major effort to improve the ratio of deprecation messages. In C#, the value increases 22.7% (from 55.1% to 77.8%), showing more concerns of developers with deprecated elements. In Java, 23% of the systems increase the number of replacement messages, while only 18.6% decrease. In C#, 20.1% of the systems increase this per-

centage, and 10% decrease. Considering several releases, we note that most systems increase and decrease the percentage of replacement messages over time.

4.3 RQ3. What are the characteristics of software systems with high and low frequency of replacement messages?

In this research question, we investigate whether system popularity, size, community, activity, and maturity have an impact on the way developers deprecate API elements, as indroduced in Section 3.4. We perform this investigation by comparing *top* and *bottom* systems; *top* systems have 100% of their API elements deprecated with replacement messages, while *bottom* barely do that.

Table 4.3 presents the metrics and their respective *p-values* and *d* applied on *top* and *bottom* systems, both for Java and C# (see Section 3.5). Metrics in bold have *p-value* < 0.05, and $d > 0.147$, *i.e.,* they are statistically significant different with at least a small effect size in *top* and *bottom* systems.

For Java, the selected *top* and *bottom* systems are statistically significant different with at least a small effect size in 7 out of the 12 metrics, including all size and activity metrics as well as number of contributors and average files per contributor in community. The effect size is large in one metric (number of commits), medium in three (number of files, number API elements, and number of contributors), and small in three (number of contributors, average files per contributor, and average days per releases). Regarding the C# systems, eight metrics present statistical significance: all popularity and size metrics, and number of contributors, number of commits, and number of releases. The effect size is large in one metric (number of contributors), medium in six (all popularity and size metrics, and number of commits), and small in one (number of releases).

In the following we investigate each dimension.

- **Popularity.** In the Java systems, we detect that there is no difference in *top* and *bottom* systems with respect to the popularity metrics. In contrast, for C# systems, we note that all popularity metrics have statistical difference. According to the relationship column, popular C# systems tend to provide less replacement messages in deprecated APIs.

Table 4.3. Metrics and their respective *p-values* and *d* on *top* and *bottom* systems. Bold values mean *p-value* < 0.05 (statistically significant different), and $d > 0.147$ (at least a small effect size). Level of significance for d-values: L = large, M = medium, S = Small, N = negligible. Rel. = relationship: "+" = *top* systems have significantly higher value on this metric. "-" = *bottom* systems have significantly higher value on this metric.

Dimension	Metric	Java			C#		
		p-value	d-value	Rel.	p-value	d-value	Rel.
Popularity	number of stars	0.674	0.142 (N)	+	**0.004**	**0.407 (M)**	-
	number of watchers	0.018	0.04 (N)	+	**<0.001**	**0.454 (M)**	-
	number of forks	0.028	0.08 (N)	+	**0.003**	**0.447 (M)**	-
Size	number of files	**<0.001**	**0.459 (M)**	-	**0.034**	**0.419 (M)**	-
	number of API elements	**<0.001**	**0.374 (M)**	-	**<0.001**	**0.414 (M)**	-
Community	number of contributors	**<0.001**	**0.376 (M)**	-	**0.009**	**0.537 (L)**	-
	avg. files per contrib.	**<0.001**	**0.227 (S)**	-	0.648	0.160 (N)	-
	avg. API elem. per contrib.	<0.001	0.123 (N)	-	0.303	0.110 (N)	-
Activity	number of commits	**<0.001**	**0.563 (L)**	-	**0.009**	**0.342 (M)**	-
	number of releases	**0.001**	**0.206 (S)**	-	**0.014**	**0.244 (S)**	-
	avg. days per release	**0.004**	**0.182 (S)**	-	0.201	0.292 (S)	-
Maturity	age (in number of days)	0.253	0.009 (S)	+	0.102	0.363 (M)	-

- **Size.** For Java and C#, we observe that *top* systems are smaller than *bottom* ones, as measured both in *number of files* and *number of API elements* (notice the "-" on the relationship column). In fact, it is intuitive to consider that smaller systems are easier to maintain and to keep track of API elements, which also facilitates the provision of replacement messages.

- **Community.** We can see that *top* systems have less contributors than *bottom* ones, in both Java and C#. This result is somehow related to the previous one: it is expected that smaller systems have less contributors. When we check the ratio of *files per contributor*, we notice that relative numbers are significant in Java systems. On other words, Java systems with few contributors and API files per contributor are more likely to have replacement messages. For C#, *top* systems

have fewer contributors than *bottom* ones. In opposite to Java, the ratio of *files per contributor* has no significance. In both languages, the ratio of *API elements per contributor* also does not have statistical significance.

- **Activity.** For the activity dimension, we observe that the *top* systems in Java have less commits and releases than *bottom* ones. An explanation is that *bottom* systems have more code changes, thus they may be more likely to degrade their APIs. Like Java, *number of commits* and *number of releases* impact in the way developers deprecate API elements in Java. We also note that Java systems released in small time deprecate their APIs with replacement messages.

- **Maturity.** For Java and C#, we could not find relevant differences between *top* and *bottom* systems with respect to their maturity (*i.e.,* age in number of days). Although, someone might expect older systems to be more stable and to provide better APIs, in fact we concluded that system age has no effect on the way developers deprecate API elements with replacement messages.

As examples, we present in Tables 4.4 and 4.5 comparisons between *top* and *bottom* systems, for Java and C#, respectively. For Java, the *top* system is xiprox-/ErrorView: it has 100% of its API elements deprecated with replacement messages which corresponds to three elements. The *bottom* one is apache/maven: it has 24% (48 from a total of 199) of its API elements deprecated with replacement messages. In fact, size, community and activity aspects of both systems are clearly distinct: xiprox-/ErrorView is easier to manage when comparing to apache/maven. For C#, the *top* system is GitTools/GitLink: it has 100% of its API elements deprecated with replacement messages. The *bottom* one is AutoFixture/AutoFixture: it has 20% (30 out of 146) of its API elements deprecated with replacement messages. Therefore, we can note that size, community and activity aspects of both systems are divergent: AutoFixture/AutoFixture tends to be easier to administer than GitTools/GitLink.

Summary: For Java, top systems are statistically different from bottom ones in 7 out of 12 metrics. For C#, we found significance in 8 metrics. Top systems tend to be smaller in terms of number of files and API elements, but have more contributors per files. For C#, popularity impacts the way developers deprecate their APIs, while for Java this metric has no significance. In both languages, system maturity has no effect on the way developers deprecate API elements with replacement messages.

Table 4.4. Comparison between a top and bottom Java system.

Metric	median values	XIPROX/ERRORVIEW (top system)	APACHE/MAVEN (bottom system)
number of files	668	67	1,934
number of API elem	4,901	101	8,601
number of contrib	18	2	45
avg. files per contrib	37.3	23.5	42.9
number of releases	25	4	44
number of commits	1,276	47	10,070
avg. days per release	54	42.5	99.3

Table 4.5. Comparison between a top and bottom C# system.

Metric	median values	GITTOOLS/GITLINK (top system)	AUTOFIXTURE/AUTOFIXTURE (bottom system)
number of stars	300	201	812
number of watchers	66	18	84
number of forks	119	27	143
number of files	35,543	7,123	72,029
number of API elem	4,270	242	6,257
number of contrib	22	19	38
number of commits	1,133	183	2,973
number of releases	19	8	175

4.4 Threats to Validity

We organize threats to validity in the following three categories:

4.4.1 Construct Validity

The construct validity is related to whether the measurement in the study reflects real-world situations.

Classification of deprecation messages. One threat of our study is that deprecated API elements may be incorrectly classified as having or not having replacement messages. In order to assess this threat, we performed two analyses in Java and C# systems. For Java, we manually analysed 500 randomly selected deprecation messages classified as denoting replacement messages. We detected 4 false-positives (<1%), *i.e.,* we classified the messages as replacement messages but they are not. Listing 4.1 shows an example of false-positive classification. In this example, the message used in the Javadoc *@deprecated* tag (line 2) was classified as replacement message, but it in fact

does not suggest alternatives to the deprecated field. In this case the message contains the keyword *use*, but does not suggest any replacement to the deprecated field.

```
1 /**
2 * @deprecated Deprecated, do not use.
3 */
4 public static final int CU_DEVICE_ATTRIBUTE_CAN_TEX2D_GATHER = 44;
```

Listing 4.1. Example of false-positive message in Java - deeplearning4j/nd4j

Next, we manually analysed 500 randomly selected deprecation messages classified as not replacement messages. In this case, we detected 26 (5%) false-negatives, *i.e.,* the message were classified as not including replacement information, but they indeed do include. Listing 4.2 presents an example of false-negative classification. In this example, the message used in the Javadoc *@deprecated* tag (line 3) was incorrectly classified as not including a replacement message, but it clearly includes. The message does not contain the keywords used to define replacement messages, but suggest the use of method *getExceptions* instead of deprecate method *exceptionIterator*.

```
1 /**
2 * @return The exception iterator.
3 * @deprecated in favor of #getExceptions
4 */
5 @Deprecated
6 public Iterator<Throwable> exceptionIterator() {
7     //...
8 }
```

Listing 4.2. Example of false-negative message in Java - spring-projects/spring-integration

For C#, we perform the same analysis. We found 1 (<1%) false-positive replacement message which is presented in Listing 4.3. In this example, the message used in the Obsolete attribute (line 1) to deprecate the method *GetMatch* was classified incorrectly as replacement message. The message contains the keyword *use*, but not contains suggestion for replacement.

```
1 [Obsolete("do not use this method", true)]
2 public static bool GetMatch(CallSite site) {
3     //...
4 }
```

Listing 4.3. Example of false-positive message in C# - mono/mono

We found also 15 (3%) false-negatives. Listing 4.4 presents an example of false-negative classification. In this example, the message used in the Obsolete attribute (line 1) was not classified as a replacement message, but it indeed documents that the property was moved to *AuthFeature.MaxLoginAttempts*. The message does not contain

the keywords used to indicate replacement messages, but informs that the property was moved.

```
1 [Obsolete("Moved to AuthFeature.MaxLoginAttempts")]
2 public int? MaxLoginAttempts
3 {
4     get //...
5     private set //...
6 }
```

Listing 4.4. Example of false-negative message in C# - SERVICESTACK/SERVICESTACK

Therefore, in both cases (false-positives and false-negatives), the risk of wrong classification is low, so this threat is reduced.

4.4.2 Internal Validity

The internal validity is related to uncontrolled aspects that may affect the experimental results.

Findings Validation. We paid special attention to the appropriate use of statistical machinery (*i.e.,* Mann-Whitney test, Cliff's Delta effect size) when reporting our results in Research Question 3. This reduces the possibility that such results are due to chance.

Correlation is not Causation. In Research Question 3, we examined whether there are metrics associated with top and bottom systems. Notice, however, that correlation does not imply causation [Couto et al., 2014]. Thus, more advanced statistical analysis, *e.g.,* causal analysis [Retherford and Choe, 2011], can be adopted to further extend our analysis.

Java Parser Implementation. A possible threat is the possibility of errors in the implementation of our AST parser, which detects deprecated API elements. However, because this implementation is based on JDT (a library developed by Eclipse), the risk of this threat is reduced.

C# In-house Tool Implementation. Another possible threat is the possibility of errors in our C# in-house tool to identify deprecated elements. In order to fix this threat, we manually analysed the precision and recall of the tool in three systems: FACEBOOK-CSHARP-SDK/FACEBOOK-CSHARP-SDK (26 deprecated elements), ANTARIS/RAZORENGINE (62 deprecated elements) and AZURE/AZURE-STORAGE-NET (107 deprecated elements). For the three analysed systems, the tool founded all deprecated elements. Thus, the risk of this threat is also low.

4.4.3 External Validity

The external validity is related to the possibility to generalize our results. We focused on the analysis on 622 Java and 229 C# open-source systems. Therefore, they are credible and representative case studies. Such systems are hosted in GitHub, the most popular code repository nowadays. Despite these observations, our findings cannot be directly generalized to other systems, specifically to systems implemented in other programming languages or commercial ones.

4.5 Final Remarks

In this section we summarize our findings regarding the three presented research questions. First, we found that 66.7% of the API elements in Java are deprecated with replacement messages. This percentage is 77.8% for C#. We also concluded that developers are more concerned to provide replacement messages to types than fields for Java systems. The percentage of replacement messages for types is 71.4%, while for fields is 50%. However, in C# the percentage is the same for all API elements (75%). Finally, we concluded that this percentage is greater in libraries than non-libraries. In Java, the value is 71.2% for libraries and 60% for non-libraries. In C#, these values are 81.8% and 73.7%, respectively.

In RQ #2, we investigate the impact of software evolution on the frequency of replacement messages. The relative number of replacement messages increases more in C# than in Java. The increase between the first and last considered releases in Java is 8%, while for C# is 22.7%. Therefore, we note a major effort in C# developers to provide replacement messages than in Java. We also note that systems with more releases are more likely to change their API elements, consequently, they show a variation in their percentage of replacement messages.

Finally, in RQ #3, we investigate the characteristics that have impact on the way developers deprecate APIs. For Java, we founded statistical significance in all size metrics, all activity metrics, *number of contributors*, and *average files per contributor*. The significant metrics for C# are all popularity metrics, all size metrics, *number of contributors*, *number of commits*, and *number of releases*. In both languages, maturity has no impact on the way of developers use replacement messages.

Chapter 5

Practical Implications

In this chapter, we discuss possible practical implications of the main study described in this master dissertation. In Section 5.1, we motivate a possible application for this study. Section 5.2 presents our study design. Section 5.3 shows our preliminary feasibility results. Finally, Section 5.4 presents concluding remarks.

5.1 Motivation

In the study described in Chapters 3 and 4, we note that:

- API elements are usually deprecated without replacement message (33.3% per system in Java and 22.2% in C#, on the median);
- In Java, the percentage of deprecated API elements with replacement messages barely increases over time (from 58.7% to 66.7%). For C#, the median per system increases from 55.1% to 77.8%;
- The relative number of replacement messages increases in only 19.4% of the Java systems. For C#, this proportion is 20.1%. This means that there is no major effort to improve replacement messages in most systems.

Consequently, it might be possible to design and implement a recommendation tool that automatically infers replacement messages by mining real solutions adopted by developers. More specifically, even when there is no explicit replacement message, API clients may take their own decisions to replace a deprecated API element by another one. Therefore, a recommendation tool could be designed to learn such decisions automatically, particularly when a common decision is followed by many clients.

For example, suppose that type T from an API A is deprecated without a replacement message. Consider also that $C_1, C_2, ..., C_n$ are clients of the API A that reference to the deprecated type T. In this context, suppose also that along their version history most clients C_i replaced the references from T to another type T'. Therefore, in this case a recommendation tool can suggest that a deprecated message like "use type T' instead" should be added to the declaration of T, as illustrated in Figure 5.1.

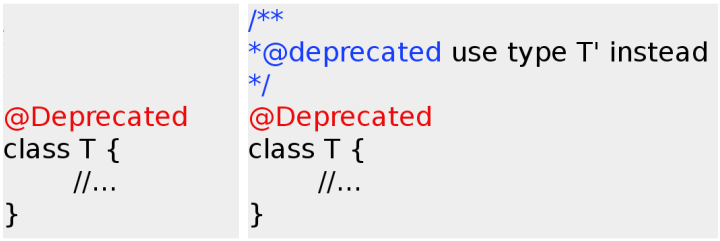

Figure 5.1. Deprecated type T without replacement message on the left and an example of suggested replacement message on the right.

In this chapter, we investigate the feasibility of designing and implementing such a tool. To this purpose, we rely on data provided by APIWAVE, which is a tool proposed by Hora and Valente [2015] to assist client developers on evolving their systems to newer or improved APIs. To support this activity, APIWAVE provides data about API migration at type level, which is mined from the textual difference between two versions of a type. More specifically, by mining the import statements of these versions, the tool infers that a type T was replaced by a type T'. The current version of APIWAVE includes data about the evolution of top-1,000 most popular GitHub Java projects, from which 320K frameworks/libraries, packages, and types are extracted. Finally, the tool provides a ranking with the most commons API migrations, which is used to support the feasibility study described in this chapter.

5.2 Study Design

5.2.1 Dataset

In order to support the proposed study, we collect the top-3,000 API migrations provided by APIWAVE. This data is organized as evolution rules in the following format: $T \rightarrow T'$. This rule expresses that type T (left side) is commonly replaced by type T' (right side), in the 1,000 GitHub projects mined by APIWAVE. Table 5.1 shows several examples of evolution rules, including both their left and right sides.

Table 5.1. Examples of evolution rules

Left Side	Right Side
junit.framework.Assert	org.junit.Assert
org.neo4j.helpers.Function	org.neo4j.function.Function
org.hibernate.criterion.Expression	org.hibernate.criterion.Restrictions
com.sk89q.worldedit.data.DataException	com.sk89q.worldedit.world.DataException
com.alibaba.dubbo.rpc.RpcConstants	com.alibaba.dubbo.common.Constants
org.sonar.api.BatchComponent	org.sonar.api.BatchSide

As presented in Table 5.1, some popular rules detected by APIWAVE are:

- $junit.framework.Assert \rightarrow org.junit.Assert$. This is the most popular rule in APIWAVE dataset. This migration occurred in JUNIT-TEAM/JUNIT, on version 4.0. The type *Assert* was moved to package $org.junit$;
- $org.neo4j.helpers.Function \rightarrow org.neo4j.function.Function$. This migration occurred in NEO4J/NEO4J, on version 2.3.3. The type $Function$ was moved to $org.neo4j.function$;
- $org.sonar.api.BatchComponent \rightarrow org.sonar.api.BatchSide$. This migration happened on version 5.2 of SonarSource/sonarqube. The new type improves some functions implemented by the deprecated one.

Among the top-3,000 evolution rules mined by APIWAVE, we found 720 rules where their left side correspond to a type available in our original dataset of 622 Java systems (considered in the study described in Chapters 3 and 4). These are exactly the types investigated in this new study. For example, the APIWAVE dataset includes the following rule: $org.neo4j.helpers.Function \rightarrow org.neo4j.function.Function$, and our dataset includes the type $org.neo4j.helpers.Function$, which matches the left side of this rule.

Considering these 720 rules, there are 44 rules whose left side is a deprecated type. For example, the APIWAVE dataset includes the following rule: $org.apache.commons.logging.Log \rightarrow org.slf4j.Logger$, where $org.apache.commons.logging.Log$ is a deprecated type.

In other words, we do not consider types that are not part of our original dataset of 622 Java systems. In fact, APIWAVE may produce rules not related to API deprecation. For example, the rule $java.util.List \rightarrow java.util.Collection$ is discarded because $java.util.List$ is not in our dataset, and it is clearly not in the context of API deprecation.

Considering the 44 evolution rules where the left side is a deprecated type, we found that 32 types have a replacement message, while 12 types do not have

such messages. For example, the right side of the evolution rule *org.hibernate-.criterion.Expression* → *org.hibernate.criterion.Restrictions* corresponds exactly to the replacement message founded in *org.hibernate.criterion.Expression*, as shown in Listing 5.1 (line 3).

```
1 /**
2  * Factory for Criterion objects.  Deprecated!
3  * @deprecated Use {@link Restrictions} instead
4  */
5 @Deprecated
6 public final class Expression extends Restrictions {
7     //...
8 }
```

Listing 5.1. Example of replacement message (line 3) that matches an evolution rule inferred by APIWAVE - HIBERNATE/HIBERNATE-ORM

5.2.2 Research Questions

In this study, we assess whether a recommendation tool would be worth to design and implement. Thus, we investigate two research questions:

RQ #1. What is the precision of a tool for recommending replacement messages?
As commonly adopted in the literature, we calculate precision as:

$$Precision = \frac{TP}{TP + FP}$$

where,

- **TP (True Positive)**: when a recommendation provided by APIWAVE matches the replacement message from a deprecated type.
- **FP (False Positive)**: when a recommendation provided by APIWAVE does not match the replacement message from a deprecated type.

RQ #2. What is the recall of a tool for recommending replacement messages?
As commonly adopted in the literature, we calculate recall as:

$$Recall = \frac{TP}{TP + FN}$$

where,

- **FN (False Negative)**: when a replacement message from a deprecated type is not covered by APIWAVE data.

5.3 Results

RQ #1. What is the precision of a tool for recommending replacement messages?

We compute a precision of 73% for the recommendations provided by APIWAVE: 32 out of 44 recommendations are true positives. As an example of true positive, we present the *junit.framework.Assert* case. The rule *junit.framework.Assert* → *org.junit.Assert* is the most popular, as mined by APIWAVE. The replacement message for this type matches the rule mined by APIWAVE, as shown in Listing 5.6 (line 4).

```
1 /**
2  * A set of assert methods.  Messages are only displayed when an assert fails.
3  *
4  * @deprecated Please use {@link org.junit.Assert} instead.
5  */
6 @Deprecated
7 public class Assert {
8     //...
9 }
```

Listing 5.2. True positive example - JUNIT-TEAM/JUNIT

As an example of false positive, we present the *android.support.v7.app.ActionBarActivity* case. APIWAVE shows that the rule *android.support.v7.app.ActionBarActivity* → *android.app.Activity* is also a popular migration. However, the real replacement message for this type does not match the rule inferred by APIWAVE, as shown in Listing 5.3 (line 2).

```
1 /**
2  * @deprecated Use {@link android.support.v7.app.AppCompatActivity} instead.
3  */
4 @Deprecated
5 public class ActionBarActivity extends AppCompatActivity {
6     //...
7 }
```

Listing 5.3. False positive example - ANDROID/PLATFORM_FRAMEWORKS-_SUPPORT

As we can see in Listing 5.3, the developers who deprecated this type are suggesting the use of the alternative type *android.support.v7.app.AppCompatActivity*, while APIWAVE suggests to use the *android.app.Activity* type.

RQ #2. What is the recall of a tool for recommending replacement messages?

In order to compute the recall, we selected three popular systems with deprecated types covered by APIWAVE: SONARSOURCE/SONARQUBE, JUNIT-TEAM/JUNIT, and GOOGLE/GUAVA.

For **SONARSOURCE/SONARQUBE**, the recall is 28.2% for the recommendations provided by APIWAVE (APIWAVE provides recommendations for 13 out of 46 replacement messages). In this context, we present examples of true positives as well as false negatives.

As an example of true positive, we show the *org.sonar.api.BatchComponent* case. The rule *org.sonar.api.BatchComponent* → *org.sonar.api.BatchSide* is the most popular one for the *org.sonar.api.BatchComponent* type. The replacement message for this type matches this rule, as presented in Listing 5.4 (line 3).

```
1 /**
2  * @since 2.2
3  * @deprecated since 5.2 use {@link BatchSide} annotation
4  */
5 @Deprecated
6 public interface BatchComponent {
7     //...
8 }
```

Listing 5.4. Example of true positive in SONARSOURCE/SONARQUBE

As an example of false negative, we present the *org.sonar.server.paging.Paging* case (Listing 5.5, line 2). For this type, APIWAVE did not find any rules.

```
1 /**
2  * @deprecated use {@link org.sonar.server.search.Result}
3  */
4 @Deprecated
5 public class Paging {
6     //..
7 }
```

Listing 5.5. Example of false negative in SONARSOURCE/SONARQUBE

For **JUNIT-TEAM/JUNIT**, the recall is 30.7% for the recommendations provided by APIWAVE (APIWAVE provides recommendations for 4 out of 13 replacement messages).

As an example of true positive, we analyse the *org.junit.matchers-.JUnitMatchers* case. The rule *org.junit.matchers.JUnitMatchers* → *org.hamcrest.junit.JUnitMatchers* is the most popular one for the *org.junit.matchers-*

```
1 /**
2  * @deprecated use {@code org.hamcrest.junit.JUnitMatchers}
3  */
4 @Deprecated
5 public class JUnitMatchers {
6     //...
7 }
```

Listing 5.6. Example of true positive in JUNIT-TEAM/JUNIT

.JUnitMatchers type. The replacement message for this type matches this rule , as presented in Listing 5.6 (line 2).

As an example of false negative, we show the *org.junit.internal.matchers.ThrowableCauseMatcher* case, presented in Listing 5.9 (line 2). For this type, APIWAVE did not find any rule.

```
1 /**
2  * @deprecated use {@code org.hamcrest.junit.ExpectedException}
3  */
4 @Deprecated
5 public class ThrowableCauseMatcher<T extends Throwable> {
6     //...
7 }
```

Listing 5.7. Example of false negative in JUNIT-TEAM/JUNIT

Finally, for **GOOGLE/GUAVA**, the recall is 37.5% (APIWAVE provides recommendations for 3 out of 8 replacement messages).

As an example of true positive, we see the *com.google.common.base.Objects.ToStringHelper* case. The rule *com.google.common.base.Objects.ToStringHelper* → *com.google.common.base.MoreObjects.ToStringHelper* is the most popular one. The rule matches the message, as presented in Listing 5.8 (line 3).

```
1 /**
2    * @deprecated Use {@link MoreObjects.ToStringHelper} instead.
3    */
4   @Deprecated
5   public static final class ToStringHelper {
6     //...
7   }
8 }
```

Listing 5.8. Example of true positive in GOOGLE/GUAVA

As an example of false negative, presented in Listing 5.9 (line 3), we show the *com.google.common.collect.MapConstraint* case. For this type, APIWAVE did not find rules.

The application scenario presented in section shows promising results, with good precision and recall. Notice, however, that the not so high recall can be justified by

```
1 /**
2  * @since 3.0
3  * @deprecated Use {@link Preconditions} for basic checks.
4  */
5 @Deprecated
6 public interface MapConstraint<K, V> {
7     //..
8 }
```

Listing 5.9. Example of false negative in GOOGLE/GUAVA

the heuristic and dataset used by APIWAVE. More specifically, when comparing two versions of a class, APIWAVE extracts evolution rules from cases where only one type is removed and only one type is added in the import statements. The positive side of this approach is that it is more likely to obtain rules with higher precision. However, this comes at the cost of producing fewer rules (*i.e.,* lower recall). Moreover, APIWAVE mines a limited number of client projects, thus, naturally, it may miss some evolution rules. Future studies may adopt different heuristic and increase the dataset in order to produce higher recall.

5.4 Final Remarks

In this chapter, we investigated practical implications of the study presented in this dissertation. Initially, we described the motivation of this new study and we motivated the possibility of the implementing a tool to recommend replacement messages for deprecated API elements. Next, we explained the study design, where we proposed two research questions to assess precision and recall of such tool. We computed a precision of 73% for recommendations provided by APIWAVE. For recall, we computed this value for three popular systems. Recall, for SONARSOURCE/SONARQUBE is 28%; for JUNIT-TEAM/JUNIT and GOOGLE/GUAVA the values for recall are 30.7% and 37.5%, respectively. These numbers show promising results, suggesting that a recommendation tool for elements deprecated without messages is indeed possible to be created, with good precision. Future work may focus, for example, on increasing recall.

Chapter 6

Conclusion

We organise this chapter as follows. First, Section 6.1 provides a brief summary of the dissertation. Section 6.2 discusses related work. Next, Section 6.3 reviews the contributions of our research. Finally, Section 6.4 suggests further work.

6.1 Overview

First, this dissertation presented a large scale empirical study about the adoption of replacement messages on deprecated API elements. We focused on three major questions: (i) the frequency of deprecated API elements with replacement messages, (ii) the impact of software evolution on such frequency, and (iii) the characteristics of systems correctly deprecating API elements. The study was performed in the context of 622 Java and 229 C# popular and real-world systems. We reiterate next the most interesting findings from this first study:

- 66.7% of the API elements in Java are deprecated with replacement messages per system (on the median). For C#, this value is 77.8%.

- The percentage of deprecated API elements with replacement messages does not improve over time in Java systems. By contrast, for C#, we note a considerable adoption of replacement messages.

- Systems that deprecate API elements in a correct way tend to be smaller and they have proportionally fewer contributors. In contrast, system maturity has no impact. For C#, we identify that popularity has a negative influence on the percentage of messages.

In addition to the main study, we perform a second study in order to evaluate the feasibility of a recommendation tool designed to infer replacement messages by mining solutions adopted by developers. In this study, we compute the precision and recall of the proposed tool. For precision, we note that 73% of the recommendations provided by the tool are true positives, *i.e.,* correspond to the real replacement messages of deprecated API elements. We compute recall for three relevant systems, *i.e.,* the percentage of replacement messages covered by the tool. The computed recall measures are 28.2% (SONARSOURCE/SONARQUBE), 30.7% (JUNIT-TEAM/JUNIT), and 37.5% (GOOGLE/GUAVA). These results suggest that the proposed recommendation tool can be provide real value to software developers.

6.2 Related Work

We separate related work in two categories, the first one about the impact of API evolution and second one in the context of API evolution analysis.

6.2.1 API Evolution Impact

McDonnell et al. [2013] investigate API stability and adoption on a small-scale Android ecosystem. The authors found that Android APIs are evolving fast and client adoption is not following the evolution pace. Also in the Android context, Linares-Vásquez et al. [2014] analyze how API changes trigger questions and activity in StackOverflow. Their results suggest that Android developers normally have more questions when the API behavior is modified.

In a large-scale study, Robbes et al. [2012] investigate the impact of API deprecation in an ecosystem, written in the dynamic typed programming language Smalltalk. They detected that some deprecations have large impact on the ecosystem and that the quality of deprecation messages should be improved. The authors provide evidence that APIs are usually deprecated with missing and unclear messages. However, their focus is on impact analysis, so they do not conduct on in-depth investigation deprecation messages, as we performed in this dissertation.

In a recent work, Hora et al. [2015] studied the impact of API replacement and improvement (*i.e.,* not API deprecation) on a large-scale ecosystem also written in Smalltalk. The results of this study confirm the large impact on client systems, and hints that deprecation mechanisms should be more adopted.

6.2.2 API Evolution Analysis

Several approaches were proposed to support API evolution and reduce the developers' efforts. Henkel and Diwan [2005] propose CatchUp, a tool that uses a modified IDE to capture and replay refactorings related to API evolution. Chow and Notkin [1996] present an approach that is supported by API developers: they annotate changed methods with replacement rules that will be used to update client systems. Hora et al. [2014]; Hora and Valente [2015] propose APIEvolutionMiner and APIWAVE, which are tools to support keeping track of API evolution and popularity.

Kim et al. [2007] propose a tool to automatically infer rules from structural changes, computed from modifications at or above the level of method signature. Kim and Notkin [2009] propose LSDiff, a tool to support computing differences between two versions of one system. In this case, the authors take into account the body of the method to infer rules, improving their previous work [Kim et al., 2007]. Nguyen et al. [2010] propose LibSync, a tool that uses graph-based techniques to help developers migrate from one framework version to another. Dig and Johnson [2005] support developers to better understand the requirements for migration tools. For instance, they found that 80% of the changes that break client systems are refactorings.

Dagenais and Robillard [2008] present SemDiff, a tool that suggests replacements for API elements based on how they adapt themselves to these changes. Schäfer et al. [2008] propose to mine API usage change rules from client systems. Wu et al. [2010] present AURA, an approach that combines call dependency and text similarity analyses to produce evolution rules. Meng et al. [2012] propose a history-based matching approach (named HiMa) to support framework evolution. In this case, rules are extracted from the revisions in code history together with comments recorded in the evolution history of the framework. Hora et al. [2012, 2013] focus on the extraction of API evolution rules that only make sense for a system or domain under analysis. Using association rule mining algorithms, Borges and Valente [2015] mine 1,952 usage patterns, from a set of 396 Android applications. They report that the Android API has many undocumented and non-trivial usage patterns.

Finally, studies also address the problem of discovering the mapping of APIs between different platforms that evolved separately. For example, Zhong et al. [2010] focus on the mapping between Java and C# APIs while Gokhale et al. [2013] study the mapping between JavaME and Android APIs.

Summary: The aforementioned studies are intended to better understand API evolution and to propose solutions to API migration. None of them, study API evolution in the context of API deprecation and their replacement messages.

6.3 Contributions

The contribution of this master dissertation are as follows:

- A large-scale empirical study about the adoption of replacement messages in Java and C# systems. In this study, we analysed the frequency of deprecated API elements with replacement messages, the impact of API evolution in this frequency, and the characteristics of systems that deprecate their elements in a correct way.
- We also provide preliminary evidence on the feasibility of designing a tool to recommend replacement messages to deprecated API elements. This tool will infer messages from the reactions of clients, *i.e.,* the solutions they followed to replace deprecated elements.

6.4 Future Work

In this dissertation, we presented a study about adoption of replacement messages in Java and C# systems. This study can be complemented with the following future work:

- We can extend our analysis to other programming languages with popular projects in GitHub. For example, we can compare systems implemented in statically and dynamically typed languages;
- We can propose a categorization of the systems under analysis in different domains to reveal and understand differences regarding API deprecation;
- As previously discussed, we can design and implement a recommendation tool to assist client developers by automatically inferring missing replacement messages in deprecated API elements.

Bibliography

Borges, H. and Valente, M. T. (2015). Mining usage patterns for the Android API. *PeerJ Computer Science*, 1(1):1--13.

Brito, G., Hora, A., and Valente, M. T. (2015). Um estudo sobre a utilização de mensagens de depreciação de APIs. In *III Workshop de Visualização, Evolução e Manutenção de Software (VEM)*.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016). Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Chow, K. and Notkin, D. (1996). Semi-automatic update of applications in response to library changes. In *International Conference on Software Maintenance*, pages 359–368.

Couto, C., Pires, P., Valente, M. T., Bigonha, R., and Anquetil, N. (2014). Predicting software defects with causality tests. *Journal of Systems and Software*, 93.

Dagenais, B. and Robillard, M. P. (2008). Recommending adaptive changes for framework evolution. In *International Conference on Software engineering*.

Dig, D. and Johnson, R. (2005). The role of refactorings in API evolution. In *International Conference on Software Maintenance*, pages 389–398.

Gokhale, A., Ganapathy, V., and Padmanaban, Y. (2013). Inferring likely mappings between APIs. In *International Conference on Software Engineering*, pages 82--91.

Grissom, R. and Kim, J. (2005). Effect sizes for research: A broad practical approach. *Lawrence Erlbaum Associates Publishers*.

Henkel, J. and Diwan, A. (2005). Catchup!: Capturing and replaying refactorings to support API evolution. In *International Conference on Software Engineering*, pages 274--283.

Hora, A., Anquetil, N., Ducasse, S., and Allier, S. (2012). Domain Specific Warnings: Are They Any Better? In *International Conference on Software Maintenance*.

Hora, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2013). Mining System Specific Rules from Change Patterns. In *Working Conference on Reverse Engineering*.

Hora, A., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2014). Apievolutionminer: Keeping api evolution under control. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 420--424. IEEE.

Hora, A., Robbes, R., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. (2015). How do developers react to API evolution? the Pharo ecosystem case. In *International Conference on Software Maintenance and Evolution*.

Hora, A. and Valente, M. T. (2015). apiwave: Keeping track of api popularity and migration. In *International Conference on Software Maintenance and Evolution*. http://apiwave.com.

Kim, M. and Notkin, D. (2009). Discovering and Representing Systematic Code Changes. In *International Conference on Software Engineering*, pages 309--319.

Kim, M., Notkin, D., and Grossman, D. (2007). Automatic inference of structural changes for matching across program versions. In *International Conference on Software Engineering*, ICSE '07, pages 333--343. IEEE Computer Society.

Konstantopoulos, D., Marien, J., Pinkerton, M., and Braude, E. (2009). Best principles in the design of shared software. In *International Computer Software and Applications Conference*.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2013). Api change and fault proneness: a threat to the success of android apps. In *Joint meeting on foundations of software engineering*.

Linares-Vásquez, M., Bavota, G., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2014). How do api changes trigger stack overflow discussions? a study on the android sdk. In *proceedings of the 22nd International Conference on Program Comprehension*, pages 83--94. ACM.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of API stability and adoption in the android ecosystem. In *International Conference on Software Maintenance*, pages 70--79. IEEE.

Meng, S., Wang, X., Zhang, L., and Mei, H. (2012). A history-based matching approach to identification of framework evolution. In *International Conference on Software Engineering*, pages 353–363.

Montandon, J. (2013). Documenting application programming interfaces with source code examples. Master's thesis, UFMG.

Moser, S. and Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Computer*, 29(9).

Nguyen, H. A., Nguyen, T. T., Wilson, Jr., G., Nguyen, A. T., Kim, M., and Nguyen, T. N. (2010). A graph-based approach to API usage adaptation. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 302--321.

Retherford, R. D. and Choe, M. K. (2011). *Statistical models for causal analysis*. John Wiley & Sons.

Robbes, R., Lungu, M., and Röthlisberger, D. (2012). How do developers react to API deprecation? The case of a smalltalk ecosystem. In *International Symposium on the Foundations of Software Engineering*, pages 56:1--56:11. ACM.

Schäfer, T., Jonas, J., and Mezini, M. (2008). Mining framework usage changes from instantiation code. In *International Conference on Software engineering*.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015). What are the characteristics of high-rated apps? a case study on free android applications. In *International Conference on Software Maintenance and Evolution*.

Tourwé, T. and Mens, T. (2003). Automated support for framework-based software. In *International Conference on Software Maintenance*.

Wu, W., Gueheneuc, Y.-G., Antoniol, G., and Kim, M. (2010). Aura: a hybrid approach to identify framework evolution. In *International Conference on Software Engineering*, pages 325–334.

Zhong, H., Thummalapenta, S., Xie, T., Zhang, L., and Wang, Q. (2010). Mining API mapping for language migration. In *International Conference on Software Engineering*, pages 195--204.