Minerando Mensagens de Depreciação Faltantes em APIs: Um Estudo de Caso no Ecossistema Android

Pedro Henrique de Moraes[1], **Caroline Lima**[1], **Andre Hora**[1]

[1] Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)

pedhmoraes@gmail.com, carollimaxp@gmail.com, hora@facom.ufms.br

Abstract. *To facilitate library migration, developers should deprecate APIs before any change that may impact on client systems. However, the literature reports that APIs are commonly deprecated without any message to support their clients. Therefore, in this paper, we propose an approach to recommend missing replacement messages in deprecated APIs. We assess the history version of the Android framework and 100 client systems. We found that some replacement messages can be detected by mining the clients, but alternatives solutions are common. Finally, based on our learning, we suggest improvements to the design of an API deprecation migration tool.*

Resumo. *Para facilitar a migração entre versões de bibliotecas de software, desenvolvedores devem depreciar APIs antes de qualquer alteração que possa impactar os clientes. No entanto, a literatura reporta que APIs são geralmente depreciadas sem mensagens de substituição para auxiliar os clientes. Diante desse problema, neste artigo, propõe-se uma abordagem para a recomendação de mensagens faltantes em APIs depreciadas. Analisa-se o histórico de versões do framework Android e de 100 sistemas clientes. Como resultado, detecta-se que algumas mensagens podem ser encontradas nos clientes, mas soluções alternativas são comuns. Por fim, com base no nosso aprendizado, sugere-se melhorias para o projeto de uma ferramenta de migração de APIs depreciadas.*

1. Introdução

Bibliotecas de software são utilizadas por aplicações clientes para reuso de funcionalidades e aumento de produtividade, resultando na diminuição dos custos de desenvolvimento [Moser and Nierstrasz 1996, Tourwé and Mens 2003]. Como qualquer outro sistema, bibliotecas de software também evoluem com o passar do tempo [Hora et al. 2015, Xavier et al. 2017]; naturalmente, elas são alteradas para fornecer novas funcionalidades [Brito et al. 2018]. Consequentemente, as aplicações clientes podem ser impactadas e devem migrar para se beneficiarem das novas APIs [Bogart et al. 2016]. Nesse contexto, visando facilitar a migração entre versões, *uma boa prática é depreciar APIs antes de qualquer alteração que possa impactar os clientes* [Robbes et al. 2012].

No entanto, a literatura reporta que APIs são geralmente depreciadas *sem* mensagens de substituição para auxiliar os clientes [Brito et al. , Sawant et al. 2017]. Diante desse problema, neste artigo, propõe-se uma abordagem para a recomendação de mensagens faltantes em APIs depreciadas. Especificamente, mesmo quando não existe mensagem de substituição explícita, *os clientes tomam suas próprias decisões para substituir*

uma API depreciada por outra [Brito et al.]. Desse modo, uma técnica de recomendação pode ser projetada para aprender automaticamente a partir dessas decisões, particularmente quando uma decisão é compartilhada por vários clientes. Logo, três questões de pesquisa são propostas:

- *QP1: Qual a taxa de APIs sem mensagens de depreciação atualmente?*
- *QP2: Pode-se inferir regras de evolução de código minerando os clientes?*
- *QP3: Pode-se inferir mensagens de depreciação faltantes minerando os clientes?*

Neste estudo, foca-se na análise do ecossistema Android devido a sua grande relevância na atualidade. Particularmente, para responder as questões de pesquisa, analisa-se o histórico de versões do framework Android e de 100 sistemas clientes. Desse modo, as principais contribuições deste trabalho são: (1) a proposição de uma abordagem para inferir mensagens de depreciação faltantes em APIs; (2) uma primeira avaliação dessa abordagem através da mineração de dezenas de sistemas clientes; (3) uma análise sobre o estado atual de depreciação nas APIs do Android; e (4) um conjunto de lições aprendidas e implicações para projetistas e clientes de bibliotecas.

2. Depreciação de APIs

Idealmente, *Application Programming Interfaces* (APIs), tais como classes, métodos e atributos públicos, devem ser estáveis ao evoluir. Em outras palavras: APIs não devem ser removidas, alteradas ou renomeadas *sem uma prévia comunicação aos seus clientes* [Brito et al. 2018, Xavier et al. 2017]. Essa comunicação deve ser realizada através da depreciação da API, que pode conter mensagens para ajudar os desenvolvedores (conforme apresentado na Figura 1). De fato, antes de ser removida, uma API deve primeiramente ser depreciada com uma mensagem clara para que os seus usuários (i) sejam notificados sobre a sua futura remoção e (ii) saibam como reagir a tal alteração.

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

Figure 1. API depreciada com mensagem de substituição [Brito et al.].

Infelizmente, a literatura reporta que esse processo nem sempre é seguido [Brito et al. , Sawant et al. 2017]. Ou seja, APIs são frequentemente depreciadas sem mensagens para auxiliar os seus sistemas clientes (conforme apresentado na Figura 2). Por exemplo, em um estudo em larga escala realizado anteriormente [Brito et al.], detectou-se que apenas 66,7% das APIs são depreciadas com mensagens em sistemas Java e 77,8% em sistemas C#. Isso significa que em torno de 34% das APIs analisadas em Java e 23% em C# são depreciadas sem mensagens.

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

Figure 2. API depreciada sem mensagem de substituição [Brito et al.].

3. Abordagem Proposta

Uma solução para esse problema é aprender *como* os sistemas clientes reagem quando se deparam com APIs depreciadas sem mensagens. Nesse contexto, pode-se analisar as alterações de código dos clientes em dois níveis de granularidade: (i) importações de classes e (ii) corpos de métodos. A mineração das alterações em nível de importação é uma técnica mais leve pois necessita apenas descobrir os `imports` adicionados e removidos na modificação de código. Por exemplo, a Figura 3 (esquerda) apresenta um trecho de código onde a mineração de imports pode ser utilizada para extração de informação relevante. Nesse caso, pode-se inferir que a classe `junit.framework.Assert` foi substituída pela classe `org.junit.Assert`.

```
 package org.jboss.as.modcluster;

-import junit.framework.Assert;
 import org.jboss.as.controller.PathAddress;
 import org.jboss.dmr.ModelNode;
 import org.jboss.dmr.ModelType;
+import org.junit.Assert;
 import org.junit.Test;
```

```
   public Collection<FormulaData> getChildren() {
-    List<FormulaData> result = new ArrayList<FormulaData>();
+    List<FormulaData> result = Lists.newArrayList();
     for (DecoratorContext childContext : decoratorContext.getChildren()) {
       result.add(new DefaultFormulaData(childContext));
     }
```

Figure 3. Esq: alteração na importação. Dir: alteração no corpo do método.

Já para a segunda abordagem (alterações nos corpos de métodos), é preciso uma análise mais fina do código alterado, uma vez que deve-se comparar duas versões da *Abstract Syntax Tree* (AST) para extração de informação relevante. Por exemplo, a Figura 3 (direita) apresenta um trecho de código onde a mineração do corpo do método `getChildren()` pode ser utilizada; nesse caso, pode-se inferir que o uso do construtor da classe `ArrayList` foi substituído pelo método `Lists.newArrayList()`.

A abordagem proposta neste trabalho (Figura 4) utiliza Mineração de Regras de Associação [Agarwal and Srikant 1994] para detectar ambos os tipos de alterações (*ie*, nas importações e nos corpos de métodos). A seguir são descritos as quatro etapas principais.

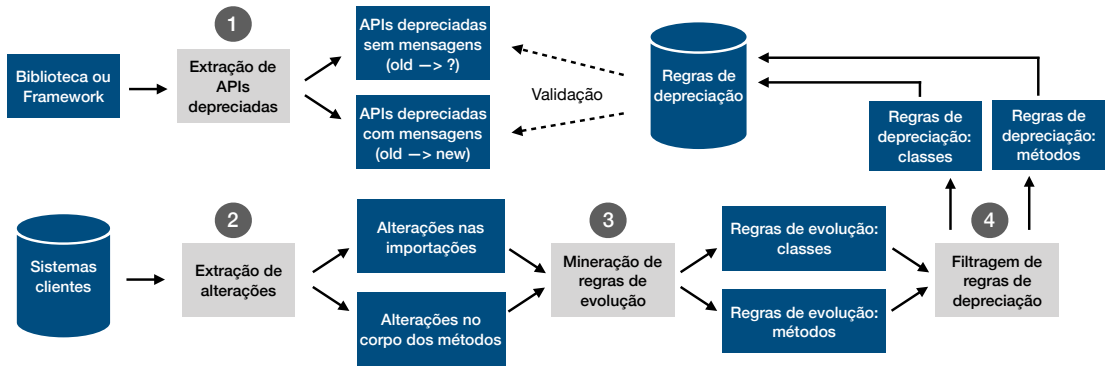

Figure 4. Abordagem para mineração de mensagens de depreciação faltantes.

1. Extração de APIs depreciadas. A primeira etapa da solução proposta consiste em extrair as APIs depreciadas de uma determinada biblioteca ou framework. Para tal, utiliza-se a ferramenta de mineração de APIs proposta por Brito et al. [Brito et al. 2016, Brito et al.] para detecção depreciação. A ferramenta adotada determina se uma API depreciada possui mensagem ou não, com base em um conjunto de regras preestabelecidas (*ex*: o Javadoc possui a tag `@deprecated` e a palavra *use*).

2. Extração de Alterações. Visando "aprender" com os sistemas clientes, para cada *commit* de um dado sistema, extrai-se: (i) as classes removidas/adicionadas nas importações e (ii) as invocações removidas/adicionadas no corpo dos métodos. Para descobrir as alterações nas *importações*, realiza-se uma análise textual entre as duas versões de uma classe de modo a detectar as importações removidas e adicionadas. Para descobrir as alterações no *corpo dos métodos*, utiliza-se a ferramenta FAMIX [Ducasse et al. 2011] para criação de modelos de código na versão anterior e posterior. Nesse caso, ao comparar duas versões de um método, extrai-se as invocações removidas e adicionadas.

3. Mineração de Regras de Evolução. Após a extração dos dados, aplica-se o algoritmo Apriori para detectar regras de associação [Agarwal and Srikant 1994] no formato `elemento-removido` $\rightarrow$ `elemento-adicionado`, indicando que o elemento na esquerda deve ser substituído pelo elemento na direita. Desse modo, com base nos dados gerados na etapa anterior, pode-se inferir regras que representam a evolução de classes e de métodos. Por exemplo, a mineração do código apresentado na Figura 3 (esquerda) pode gerar a regra: `junit.framework.Assert` $\rightarrow$ `org.junit.Assert`. Nota-se, no entanto, que nem todas as regras geradas são necessariamente no contexto de depreciação.

4. Filtragem de Regras de Depreciação. A última etapa consiste em filtrar as regras de evolução para selecionar apenas regras no contexto de depreciação de APIs. Especificamente, seleciona-se dois conjuntos de regras. No primeiro, nomeado *match parcial*, seleciona-se as regras de depreciação em que o lado esquerdo corresponde a uma API depreciada com mensagem. No segundo, nomeado *match total*, seleciona-se as regras de depreciação em que o lado esquerdo corresponde a uma API depreciada com mensagem e o lado direito corresponde a mensagem de substituição dessa API. Logo, pode-se validar a capacidade da abordagem acertar/errar APIs depreciadas com mensagens de depreciação.

4. Metodologia

4.1. Extração de APIs Depreciadas: Framework Android

Neste estudo, focou-se na análise do Android por representar um ecossistema altamente relevante. Minerou-se o Android Base e o Android Support, cujo os códigos estão espelhados no GitHub.[1] Realizou-se a extração das APIs depreciadas (Etapa 1, Seção 3) para os 345 mil *commits* do Android Base e 35 mil *commits* do Android Support. Em seguida, categorizou-se, manualmente, as APIs depreciadas em três grupos: com mensagem, com mensagem parcial e sem mensagem. Na QP1, essas categorias são explicadas e os resultados são discutidos.

4.2. Mineração de Regras de Evolução e Depreciação: Sistemas Clientes Android

Visando descobrir como os clientes lidam com APIs depreciadas, realizou-se uma primeira avaliação com sistemas clientes do Android (Etapas 2, 3 e 4, Seção 3). Foram selecionados no GitHub os 100 projetos Java mais populares baseados no Android; como critério de inclusão, selecionou-se os projetos com a palavra "Android" em suas descrição. Rodou-se o algoritmo de mineração de regras de associação com suporte 3 e confiança 50%. Esses valores foram configurados após uma avaliação manual realizada pelos autores, de forma a minimizar a quantidade de regras irrelevantes.[2] Esses resultados são detalhados na QP2 e na QP3.

[1] Projetos no GitHub: platform_frameworks_base e platform_frameworks_support

[2] Discutir os detalhes dessa avaliação está fora do escopo deste artigo.

5. Resultados

QP1: Qual a taxa de APIs sem mensagens de depreciação?

A Tabela 1 apresenta a taxa de APIs depreciadas com mensagens, parcialmente e sem mensagens. Nota-se a predominância de APIs depreciadas com mensagens completas para ajudar os clientes (57.88%). Observa-se que 30.80% das APIs depreciadas não possuem mensagens relevantes para auxiliar os clientes. 11.32% das APIs são depreciadas com mensagens parciais. Nota-se também que a grande maioria das entidades depreciadas ocorrem em nível de métodos (62%, 1,961 de 3,153 casos).

API	Total	Com Msg.	%	Parcial	%	Sem Msg.	%
Classe	63	4	6.35	7	11.11	52	82.54
Método	1,961	1,229	62.67	102	5.20	630	32.13
Atributo	1,129	592	52.43	248	21.97	289	25.60
Todas	**3,153**	**1,825**	**57.88**	**357**	**11.32**	**971**	**30.80**

Table 1. Taxa de APIs depreciadas com, parcial e sem mensagens.

Quando uma mensagem está completa, poupa tempo do cliente em busca de soluções alternativas. A Figura 5 (esquerda) apresenta um exemplo em que o método depreciado `getValidNotBefore()` pode ser substituído por `getValidNotBeforeDate()`. Entretanto, mensagens relevantes nem sempre estão presentes nas APIs depreciadas. Nessa condição, um cliente deve buscar por soluções alternativas. A Figura 5 (direita) apresenta uma exemplo em que método `onNewPicture()` está depreciado com mensagem irrelevante: "*Deprecated due to internal changes*". Além desses dois casos, mensagens de depreciação podem vir parcialmente, sugerindo locais onde procurar soluções alternativas para a depreciação. A Figura 5 (abaixo) mostra o método `getSelectedNavigationIndex()` depreciado sem mensagem direta, mas com uma *url* para orientar seus clientes.

```
/**
* @deprecated Use {@link #getValidNotBeforeDate()}
*/
@Deprecated
public String getValidNotBefore() {

    return formatDate(mValidNotBefore);
}
```

```
/**
 * @deprecated Deprecated due to internal changes.
 */
@Deprecated
public void onNewPicture(WebView view, Picture picture);
```

```
/**
* @deprecated Action bar navigation modes are deprecated and not supported by
inline
* toolbar action bars. Consider using other
* <a href="http://developer.android.com/design/patterns/navigation.html">common
* navigation patterns</a> instead.
*/
@Deprecated
public abstract int getSelectedNavigationIndex();
```

Figure 5. Esquerda: depreciação com mensagem. Direita: depreciação sem mensagem relevante. Abaixo: depreciação com mensagem parcial

Categoria	Regra de Evolução	Regra de Depreciação	Match Parcial	Match Total
Classe	121	12	11	1
Método	14	1	1	0
Todas	**135**	**13**	**12**	**1**

Table 2. Regras de evolução e depreciação de APIs.

QP2: Pode-se inferir regras de evolução de código?

A segunda coluna da Tabela 2 apresenta a quantidade de regras de evolução detectadas nos 100 sistemas Android analisados. Verifica-se um total de 135 regras geradas, onde 121 são no nível de classe (extraídas das alterações nas importações) e 14 são de métodos (extraídas das alterações nos corpos dos métodos). Logo, de fato, regras de evolução são frequentemente encontradas nos sistemas clientes do Android. Em seguida, analisa-se quais dessas regras são relacionadas a depreciação de APIs.

QP3: Pode-se inferir mensagens de depreciação faltantes minerando os clientes?

A terceira coluna da Tabela 2 apresenta a quantidade de regras no contexto de depreciação. Nota-se que 13 (10%) das 135 regras de evolução são relacionas a depreciação. Dentre essas 13 regras, 12 são no nível de classes enquanto 1 é de método. Verifica-se também que dentre as 13 regras de depreciação, 12 possuem *match parcial*, *ie*, regras em que o lado esquerdo corresponde a uma API depreciada com mensagem. Nesses casos, o lado direito das regras não batem com a sugestão da API. Por exemplo, a classe `MenuItemCompat` do Android foi depreciada sugerindo a utilização da classe `MenuItem`. No entanto, a regra de depreciação detectada foi `MenuItemCompat` → `org.telegram.ui.Views.ActionBar.ActionBarMenuItem`. Logo, nota-se que os clientes tendem a utilizar soluções alternativas como substituição das APIs depreciadas. Por fim, detectou-se apenas 1 regra com *match total*. Por exemplo, a classe `Contacts` foi depreciada sugerindo a utilização da classe `ContactsContract`. Nesse caso isolado, de fato, a regra de depreciação detectada foi `Contacts` → `ContactsContract`.

6. Discussão

APIs são comumente depreciadas sem mensagens. Foi verificado que 30.8% das APIs depreciadas no framework Android não contém mensagens para ajudar os clientes. Estudos anteriores (*eg*, [Brito et al.]), cujo o foco não é o Android, detectaram uma taxa equivalente, de 34%. Esses dados evidenciam a necessidade de uma ferramenta para ajudar os clientes.

APIs são ocasionalmente depreciadas com mensagens parciais. 11.32% das APIs depreciadas apenas contém mensagens parciais (*eg*, urls de ajuda). Desse modo, somado com a taxa de APIs sem mensagens, nota-se que em torno de 42% das APIs depreciadas não ajudam diretamente seus clientes.

Regras de evolução podem ser detectadas. Confirmando estudos anteriores (*eg*, [Wu et al. 2010, Hora et al. 2015, Hora et al. 2018a]), foi verificado que regras de evolução de APIs (principalmente em nível de classe) podem ser detectadas através da técnica de mineração de regras de associação.

Regras em nível de método são mais difíceis de serem detectadas. O maior desafio da abordagem proposta foi produzir regras em nível de método. Isso ocorre pois é necessário lidar com a resolução de tipos nos clientes. Para tal, é necessário analisar suas dependências externas. Logo, sugere-se que as dependências dos clientes (*ie*, `jars` externos) sejam incluídas para aumento na quantidade de regras geradas. Além disso, para evitar os ruídos causados alterações durante a evolução, sugere-se que refatorações sejam resolvidas para uma detecção de regras mais completa [Hora et al. 2018b].

Regras de depreciação podem ser detectadas. Foram detectadas 12 regras de depreciação com *match parcial* e apenas 1 *match total*. Apesar de baixo, esses números sugerem que regras de depreciação podem ser detectadas. De fato, nesse estudo, foram analisados apenas 100 sistemas clientes. Uma hipótese, é que para para aumentar a quantidade de regras, deve-se minerar mais clientes. Por exemplo, aumentando a quantidade de clientes em um fator de 100, teoricamente, pode-se gerar milhares de regras de depreciação com *match parcial* e uma centena com *match total*.

Soluções alternativas são adotadas pelos clientes. Os clientes utilizam soluções alternativas para lidar com a evolução de APIs depreciadas. Isso aumenta o leque de possibilidades para a substituição da API depreciada, indicando que soluções não oficiais são adotadas.

7. Ameaças à Validade

Generalização dos resultados. Este estudo analisou o framework Android e 100 sistemas clientes implementados em Java. Assim, não é possível generalizar os resultados para outras linguagens de programação nem para outros ecossistemas de software.

Resolução de tipos. A ferramenta FAMIX (para a criação de modelos de código, adotada na Etapa 3) possui algumas limitações para a geração da AST. Quando o código analisado possui dependências externas, naturalmente, o nome completo de algumas APIs não são resolvidos, o que poderia impactar na qualidade das regras geradas. Portanto, para evitar regras com nomes incompletos, esses casos foram removidos da nossa base de alterações.

8. Trabalhos Relacionados

Diversas abordagens são propostas pela literatura para analisar evolução de APIs [Henkel and Diwan 2005, Hora et al. 2015, Hora et al. 2018a, Wu et al. 2010] e estudar seus efeitos colaterais [Bogart et al. 2016, Brito et al. 2018, Xavier et al. 2017]. O contexto de depreciação de APIs também é explorado [Brito et al. , Robbes et al. 2012, Sawant et al. 2017], mostrando que os clientes sofrem com a falta de mensagens e alto tempo de reação. Esses estudos, entretanto, não propõem soluções para lidar com APIs depreciadas sem mensagens.

9. Conclusão

No melhor do nosso conhecimento, a abordagem proposta é a primeira para detectar mensagens de depreciação faltantes. Verificou-se que 42% das APIs depreciadas do Android não possuem mensagens. Através da mineração de 100 clientes, algumas regras de depreciação foram geradas. Como trabalhos futuros, pretende-se: (i) realizar uma análise em larga escala (10K clientes), (ii) minimizar o problema da resolução de tipos, (iii) analisar outros ecossistemas e (iv) avaliar outras técnicas para geração das regras.

Agradecimentos: Esta pesquisa é financiada pelo CNPq e pela CAPES.

References

Agarwal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *International Conference on Very Large Data Bases (VLDB)*.

Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). How to break an API: cost negotiation and community values in three software ecosystems. In *International Symposium on Foundations of Software Engineering (FSE)*.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018). Why and How Java Developers Break APIs. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software*.

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 *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

Ducasse, S., Anquetil, N., Bhatti, M. U., Hora, A., Laval, J., and Girba, T. (2011). MSE and FAMIX 3.0: an interexchange format and source code model family.

Henkel, J. and Diwan, A. (2005). CatchUp!: capturing and replaying refactorings to support API evolution. In *International Conference on Software Engineering (ICSE)*.

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 (ICSME)*.

Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A., and Ducasse, S. (2018a). How do developers react to API evolution? a large-scale empirical study. *Software Quality Journal*, 26.

Hora, A., Silva, D., Robbes, R., and Valente, M. T. (2018b). Assessing the threat of untracked changes in software evolution. In *International Conference on Software Engineering (ICSE)*.

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

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 (FSE)*.

Sawant, A. A., Robbes, R., and Bacchelli, A. (2017). On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering*.

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

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

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of API breaking changes: A large-scale study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.