Um Estudo sobre a Utilização de Mensagens de Depreciação de APIs

Gleison Brito, André Hora, Marco Tulio Valente

[1]ASERG Group – Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – Brasil

{gleison.brito,hora,mtov}@dcc.ufmg.br

Abstract. *When an API evolves, their clients should be updated to benefit from improved interfaces. In order to facilitate this task, APIs should always be deprecated with clear replacement messages. However, in practice, there are evidences that APIs are commonly deprecated without messages. In this paper, we study a set of questions related to the adoption of API deprecation messages. We aim (i) to measure the usage of API deprecation messages and (ii) to investigate whether a tool can be built to recommend such messages. The analysis of 14 real-world software systems shows that methods are frequently deprecated without replacement messages and this problem tends to get worse with the evolution of such systems. As a result, we provide the basis for the design of a tool to support developers on detecting such missing messages.*

Resumo. *Quando uma API evolui, seus clientes devem ser atualizados para se beneficiarem de interfaces melhores. Para facilitar a atualização dos clientes, APIs devem ser sempre depreciadas com mensagens claras. No entanto, na prática, existem evidências de que APIs são comumente depreciadas sem mensagens. Nesse artigo, estuda-se um conjunto de questões relativas ao uso de mensagens de depreciação de APIs, visando (i) mensurar a utilização dessas mensagens na prática e (ii) investigar a viabilidade da construção de uma ferramenta de recomendação dessas mensagens. A análise de 14 sistemas reais confirma que métodos são frequentemente depreciados sem mensagens, e esses números tendem a aumentar com a evolução desses sistemas. Como resultado, esboça-se a proposição de uma ferramenta para auxiliar os desenvolvedores na detecção dessas mensagens faltantes.*

1. Introdução

Sistemas de software estão em constante evolução através da adição de novas funcionalidades, correção de bugs e refatoração de código. Nesse processo, as APIs desses sistemas também estão sujeitas a mudanças. Quando essas APIs evoluem, seus clientes internos e externos também devem ser atualizados para se beneficiarem de interfaces melhores.

De modo a facilitar a atualização dos clientes, algumas boas práticas devem ser adotadas pelos desenvolvedores de APIs. Por exemplo, antes de serem removidas ou renomeadas, as APIs devem ser sempre depreciadas com mensagens claras para seus clientes de modo a manter sua compatibilidade. No entanto, na prática, existem evidências de que APIs são comumente depreciadas com mensagens obscuras ou mesmo

sem mensagens, dificultando a sua atualização por parte de clientes [Wu et al. 2010, Robbes et al. 2012, Hora et al. 2015].

Neste artigo, estuda-se um conjunto de questões relativas ao uso de mensagens de depreciação de APIs para melhor entender esse fenômeno. Tem-se como objetivo (i) mensurar a utilização de mensagens na depreciação de APIs e (ii) investigar a viabilidade da construção de uma ferramenta de recomendação dessas mensagens. Desse modo, estuda-se três questões de pesquisa centrais:

1. Com que frequência métodos são depreciados com mensagens de auxílio aos desenvolvedores? Com que frequência métodos são depreciados sem essas mensagens?
2. A quantidade de métodos depreciados sem mensagens tende a aumentar ou diminuir com passar do tempo?
3. Métodos depreciados com mensagens realmente sugerem como o desenvolvedor deve proceder?

Para responder essas questões analisa-se a evolução de 14 sistemas de software reais desenvolvidos em Java, tais como Hibernate, Apache Tomcat e Google Guava. Logo, as principais contribuições desse trabalho são: (1) apresenta-se uma caracterização da utilização de mensagens de depreciação de APIs em sistemas de software reais e (2) fornece-se uma base para recomendação dessas mensagens para desenvolvedores.

Esse artigo está organizado como descrito a seguir. A Seção 2 apresenta a metodologia proposta e a Seção 3 os resultados. Na Seção 4, são apresentadas as implicações desse estudo e na Seção 5 os riscos à validade. Finalmente, a Seção 6 discute trabalhos relacionados e a Seção 7 conclui o estudo.

2. Metodologia

Neste estudo foram analisados 14 sistemas *open-source* amplamente utilizados e escritos em Java. Para cada sistema foram analisadas três *releases*: inicial, intermediária e final (*i.e.,* última disponível). A Tabela 1 mostra a descrição dos sistemas, juntamente com as *releases* analisadas.

Em Java um método é depreciado utilizando-se a anotação *@Deprecated* e/ou a tag *@deprecated* inserida no Javadoc do respectivo método, conforme mostrado na Figura 1. Métodos depreciados através da tag *@deprecated* podem incluir mensagens em seu Javadoc sugerindo um novo método a ser utilizado através de tags *@link* e *@see*. Por outro lado, métodos depreciados por meio da anotação *@Deprecated* não possuem mensagens associadas. Nesse caso, um *warning* será emitido ao desenvolvedor pelo compilador, mas sem sugestão de como proceder.

Neste trabalho, utilizou-se a biblioteca JDT (*Java Development Tools*) para a detecção de métodos depreciados com e sem mensagens de auxílio aos desenvolvedores. Para isso, foi implementado um parser baseado em JDT para a coleta das ocorrências das anotações e das tags de depreciação em métodos, assim como suas respectivas mensagens Javadoc.

Table 1. Sistemas analisados

Sistema	Descrição	Release		
		Inicial	Inter.	Final
ElasticSearch	Servidor de buscas	1.3.7	1.4.3	1.5.2
Google Guava	Framework para manutenção de código	8.0	12.0	17.0
Netty	Framework para protocolos web	3.2.0	3.6.1	4.1.0
Apache Hadoop	Plataforma para sistemas distribuídos	1.0.4	2.4.1	2.7.0
Hibernate	Framework de persistência de dados	3.5.0	3.5.6	4.3.1
JUnit	Framework para testes	3.8.1	4.5	4.12
Apache Lucene	Biblioteca para busca textual	4.3.0	4.7.0	5.1.0
Spring Framework	Framework para aplicações web	1.0	3.0.0	4.1.0
Apache Tomcat	Servidor web	4.1.12	6.0.16	8.0.15
Log4J	Gerenciador de logs	1.0.4	1.2.13	2.2
Facebook Fresco	Gerenciador de imagens	0.1.0	0.3.0	0.5.0
Picasso	Gerenciador de imagens	1.0.0	2.0.0	2.1.1
Rhino	Implementação JavaScript em Java	1.4	1.6	1.7
Eclipse JDT	Infraestrutura para IDE do Eclipse	3.7	4.2.1	4.4.1

```java
/**
 * Does some thing in old style.
 *
 * @deprecated use {@link #new()} instead.
 */
@Deprecated
public void old() {
// ...
}
```

Figure 1. Exemplo de depreciação de método em Java

3. Resultados

3.1. Com que frequência métodos são depreciados com e sem mensagens de auxílio aos desenvolvedores?

Nesta questão de pesquisa estuda-se a frequência com que sistemas de software reais contém mensagens de depreciação de APIs. Dos 14 sistemas analisados, nove apresentam pelo menos um método depreciado sem mensagem. Esses sistemas são analisados com mais detalhes no restante do artigo.

A Tabela 2 mostra a quantidade de métodos depreciados com e sem mensagens na última *release* desses nove sistemas. Verifica-se, na prática, uma quantidade relevante de métodos depreciados sem mensagens. Por exemplo, no Apache Hadoop esse valor é de 42% dos métodos e no Netty 38%. Por outro lado, no Google Guava esse valor é de apenas 2%. Considerando todos os sistemas, dos 1,128 métodos depreciados, 278 (24%) não contém mensagens para auxiliar seus clientes.

Table 2. Número e percentagem de métodos depreciados com e sem mensagens.

Sistema	# Mét. Depreciados	# Com Mensagem	# Sem Mensagem
Elastic Search	77	55 (71%)	22 (29%)
Google Guava	168	165 (98%)	3 (2%)
Netty	80	50 (62%)	30 (38%)
Apache Hadoop	284	162 (57%)	122 (42%)
Hibernate	184	174 (95%)	10 (5%)
JUnit	29	23 (80%)	6 (20%)
Apache Lucene	79	61 (77%)	18 (23%)
Spring Framework	138	87 (63%)	51 (37%)
Apache Tomcat	89	73 (82%)	16 (18%)
Total	1128	850 (76%)	278 (24%)

3.2. A quantidade de métodos depreciados sem mensagens tende a aumentar ou diminuir com passar do tempo?

Nesta questão de pesquisa verifica-se se a quantidade de métodos depreciados sem mensagens aumenta ou diminui nas três *releases* analisadas (*i.e.,* inicial, intermediária e final).

A Tabela 3 mostra a quantidade desses métodos em cada uma das *releases*. A última coluna indica se o delta entre as *releases* foi acompanhado de um aumento (+) ou uma diminuição (–).

Table 3. Número e percentagem de métodos depreciados sem mensagens nas *releases* iniciais, intermediárias e finais.

Sistema	# Métodos Depreciados Sem Mensagens			Delta
	Inicial	Intermediária	Final	
Elastic Search	14 (36%)	30 (30%)	22 (29%)	– –
Google Guava	4 (13%)	7 (10%)	3 (2%)	– –
Netty	18 (46%)	72 (55%)	30 (38%)	+ –
Apache Hadoop	48 (30%)	102 (40%)	122 (42%)	+ +
Hibernate	0 (0%)	3 (4%)	10 (5%)	+ +
JUnit	0 (0%)	0 (0%)	6 (20%)	+
Apache Lucene	1 (3%)	12 (13%)	18 (23%)	+ +
Spring Framework	0 (0%)	6 (10%)	51 (37%)	+ +
Apache Tomcat	0 (0%)	4 (2%)	16 (18%)	+ +
Total	85 (14%)	236 (18%)	278 (24%)	+ +

Intuitivamente espera-se que a quantidade de métodos depreciados sem mensagens diminua, uma vez que as APIs desses sistemas se tornem mais robustas. No entanto, esse foi o caso de apenas três sistemas: ElasticSearch, Google Guava e Netty (entre a *release* intermediária e final). No ElasticSearch esse número caiu de 36% na *release* inicial para 29% na final. Percebe-se que a evolução do Google Guava inclui um esforço para utilização de mensagens: a quantidade de métodos sem mensagens de depreciação caiu de 13% para apenas 2%.

No entanto, a quantidade de métodos depreciados sem mensagens aumentou na maioria dos sistemas. Por exemplo, no Apache Hadoop, essa percentagem passou de 30%

na *release* inicial para 42% na final. No Hibernate, JUnit, Spring Framework e Apache Tomcat verifica-se que a percentagem inicial de 0% (*i.e.,* todos os métodos foram depreciados com mensagens). No entanto, na *release* final tem-se 5%, 20%, 37% e 18% de métodos depreciados sem mensagens, respectivamente. Considerando todos os sistemas, a percentagem subiu de 14% na verão inicial desses sistemas para 24% na final.

Uma possível explicação para a ausência de mensagens de depreciação é que os desenvolvedores podem estar adicionando mensagens na *release* imediatamente posterior. Por exemplo, eles podem em um primeiro momento depreciar o método (indicando que esse método será removido no futuro) e em um segundo momento (*i.e.,* na *release* seguinte) adicionar as mensagens. Desse modo, analisou-se as *releases inicial+1* e *intermediária+1* em busca de métodos que receberam mensagens. Especificamente, foi verificado a possibilidade de um método depreciado *somente* com a anotação @*Deprecated* receber posteriormente um Javadoc com a tag @*deprecated*. Como resultado, constatou-se que nenhum dos métodos depreciados sem mensagem foi refatorado na *release* seguinte para receber tais mensagens.

3.3. Métodos depreciados com mensagens realmente sugerem como o desenvolvedor deve proceder?

Nesta questão de pesquisa investiga-se se métodos depreciados com mensagens incluem tags (@*see* ou @*link*) ou palavras chaves (*use*) que apontem para sua substituição.

A Tabela 4 resume esses dados para a última *release* de cada sistema. No JUnit 91% dos métodos depreciados com mensagens incluem elementos para ajudar os desenvolvedores. No Google Guava, embora exista um esforço para incluir mensagens (conforme mostrado na Questão 2), verifica-se que apenas 38% das mensagens incluem esses elementos. Já no Apache Tomcat, as tags padrão @*see* e @*link* nunca são utilizadas, mas sim a palavra chave *use* em 29% das mensagens. Em suma, verifica-se uma quantidade relevante de métodos depreciados com os elementos considerados (*e.g.,* 54% utilizam a tag @*link*), no entanto, uma grande parte dessas mensagens ainda são incompletas.

Table 4. Número e percentagem de métodos depreciados com mensagens utilizando as tags *@see* e *@link* e a palavra *use*.

Sistema	# Métodos Depreciados Com Mensagens		
	@see	@link	use
Elastic Search	0 (0%)	36 (65%)	36 (65%)
Google Guava	1 (1%)	62 (38%)	62 (38%)
Netty	0 (0%)	40 (50%)	40 (50%)
Apache Hadoop	11 (7%)	136 (84%)	136 (84%)
Hibernate	4 (2%)	113 (65%)	113 (65%)
JUnit	0 (0%)	21 (91%)	21 (91%)
Apache Lucene	2 (3%)	36 (59%)	36 (59%)
Spring Framework	36 (41%)	36 (41%)	23 (26%)
Apache Tomcat	0 (0%)	0 (0%)	21 (29%)
Total	54 (6%)	480 (54%)	488 (55%)

As Figuras 2 e 3 contém trechos de código exemplificando a depreciação de métodos no sistema Spring Framework. A Figura 2 mostra a utilização da palavra chave

use acompanhado da tag @*link* indicando o método que irá substituir o depreciado. Já na Figura 3 verifica-se o uso das tags @*see* e @*link*. Nota-se que pode haver sobreposição entre as duas formas de depreciar métodos, o que justifica o fato de a soma dos totais de métodos que utilizam as tags @*see* e @*link* e a palavra chave *use* ultrapassar 100%.

```
/**
 * Merge the specified Velocity template with the given model and write
 * the result to the given Writer.
 * @param velocityEngine VelocityEngine to work with
 * @param templateLocation the location of template, relative to Velocity's resource loader path
 * @param model the Map that contains model names as keys and model objects as values
 * @param writer the Writer to write the result to
 * @throws VelocityException if the template wasn't found or rendering failed
 * @deprecated Use {@link #mergeTemplate(VelocityEngine, String, String, Map, Writer)}
 * instead, following Velocity 1.6's corresponding deprecation in its own API.
 */
@Deprecated
public static void mergeTemplate(
                VelocityEngine velocityEngine, String templateLocation, Map<String, Object> model, Writer writer)
                throws VelocityException {

        VelocityContext velocityContext = new VelocityContext(model);
        velocityEngine.mergeTemplate(templateLocation, velocityContext, writer);
}
```

Figure 2. Depreciação utilizando *use* e *@link*

```
/**
 * Determine whether the given attribute is eligible for being
 * turned into a corresponding bean property value.
 * <p>The default implementation considers any attribute as eligible,
 * except for the "id" attribute and namespace declaration attributes.
 * @param attribute the XML attribute to check
 * @see #isEligibleAttribute(String)
 * @deprecated in favour of {@link #isEligibleAttribute(org.w3c.dom.Attr, ParserContext)}
 */
@Deprecated
protected boolean isEligibleAttribute(Attr attribute) {
        return false;
}
```

Figure 3. Depreciação utilizando *@see* e *@link*

4. Implicações

Nesta seção são apresentadas duas implicações do presente estudo. A Questão 1 confirma que APIs são frequentemente depreciadas sem mensagens de substituição para ajudar os clientes. A Questão 2 mostra que esse problema se agrava com o passar do tempo: na maioria dos sistemas analisados a quantidade de APIs sem mensagens aumenta. Isso pode ser explicado pela complexidade e tamanho dos sistemas analisados. De fato, trabalhar nessas APIs pode não ser uma tarefa simples, mas sim envolver diversos desenvolvedores com os mais diversos níveis de conhecimento, sendo difícil manter consistência durante sua evolução [Wu et al. 2010, Robbes et al. 2012, Hora et al. 2015]. Nesse sentido, existe um esforço da literatura em minerar evolução de APIs (*e.g.,* [Kim and Notkin 2009, Hora et al. 2013, Meng et al. 2013, Hora et al. 2014], mas não no contexto de ajudar na depreciação de métodos. Assim, apresenta-se a seguinte implicação:

Implicação 1: Uma ferramenta de recomendação de mensagens de depreciação pode ser construída para auxiliar tanto os desenvolvedores das APIs quanto seus clientes. Essas mensagens podem ser inferidas através da mineração das reações dos clientes, *i.e.,* pode-se aprender a solução adotada pelos clientes na ausência de mensagens de substituição de métodos.

A Questão 3 mostra que uma quantidade relevante de métodos são depreciados com mensagens utilizando certas tags da linguagem Java. Quando a depreciação é realizada com uma mensagem bem estruturada, o método depreciado pode ser diretamente substituído pelo recomendado na mensagem. Essa informação fornece a base para seguinte implicação:

> *Implicação 2*: A qualidade de uma ferramenta de recomendação de mensagens de depreciação pode ser verificada através de sua correção em identificar mensagens de métodos depreciados *com* mensagens. Em outras palavras, os métodos depreciados com mensagens podem ser utilizados como um oráculo para mensurar a acurácia da ferramenta em identificar mensagens válidas.

5. Riscos à Validade

Validade Externa. Os resultados desse estudo estão restritos a análise de sistemas Java e não podem ser generalizados para outras linguagens. Adicionalmente, como os sistemas analisados são *open-source*, pode ocorrer resultados diferentes em sistemas comerciais. Além disso, o estudo descrito no presente trabalho envolveu 14 sistemas. A princípio essa quantidade poderia comprometer a generalização das conclusões obtidas. Porém, esse risco é atenuado pelo fato de os sistemas analisados serem relevantes e de alta complexidade.

Validade Interna. Neste estudo coletou-se mensagens relativas a depreciação dos métodos das APIs analisadas. Para isso foi implementado um *parser* baseado na biblioteca JDT, que é desenvolvido pelo Eclipse para lidar com AST. Logo, o fato do parser implementado ser baseado no Eclipse JDT o deixa com menos chances de erros.

Validade de Construção. Foram analisadas para cada sistema três *releases*: inicial, intermediária e final. Essas três *releases* não caracterizam todo o desenvolvimento desses sistemas. No entanto, elas caracterizam uma visão geral de suas evoluções.

6. Trabalhos Relacionados

Existem alguns trabalhos que mostram evidências de que APIs são comumente depreciadas com mensagens obscuras ou mesmo sem mensagens, dificultando o processo de atualização dos clientes [Robbes et al. 2012, Hora et al. 2015]. Esses trabalhos analisam um ecossistema com sistemas implementados em Smalltalk e concluem que a qualidade das mensagens deve ser melhorada. No presente estudo, essa análise foi estendida para sistemas Java (Questão 1). Além disso, o presente estudo aprofundou outras questões não abordadas nos trabalhos relacionados: foi investigado a evolução dos métodos depreciados sem mensagens (Questão 2) e se as mensagens realmente sugerem como os desenvolvedores devem proceder (Questão 3).

A literatura propõe diversas abordagens para lidar com evolução de APIs. Por exemplo, essa tarefa pode ser realizada com a ajuda IDEs modificadas [Henkel and Diwan 2005], a ajuda dos desenvolvedores das APIs [Chow and Notkin 1996] ou através da mineração de repositórios de software [Xing and Stroulia 2007, Kim and Notkin 2009, Wu et al. 2010, Hora et al. 2012, Hora et al. 2013, Meng et al. 2013, Hora et al. 2014, Hora et al. 2015, Hora and Valente 2015]. No entanto, nenhum desses trabalhos aborda diretamente o problema da ausência de mensagens na depreciação.

7. Conclusões

Neste trabalho caracterizou-se a utilização de mensagens de depreciação de APIs por meio da análise de 14 sistemas reais implementados em Java. Verificou-se que existe uma parcela relevante (24%) de métodos depreciados sem essas mensagens. Também foi constatado que esse problema se agrava ao longo da evolução dos sistemas analisados: na maioria dos sistemas a quantidade de métodos sem mensagens aumenta (de modo geral de 14% para 24%). A partir desses resultados verificou-se a viabilidade da construção de uma ferramenta para recomendação de mensagens de depreciação para auxiliar os desenvolvedores nas suas atividades de migração de APIs.

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

Referências

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

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

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 Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*.

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*.

Meng, N., Kim, M., and McKinley, K. S. (2013). Lase: Locating and applying systematic edits by learning from examples. In *International Conference on Software Engineering*.

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*.

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*.

Xing, Z. and Stroulia, E. (2007). Api-evolution support with diff-catchup. *Transactions on Software Engineering*, 33(12).