When Should Internal Interfaces Be Promoted to Public?

Andre Hora  
ASERG Group, DCC, UFMG, Brazil  
hora@dcc.ufmg.br  
Facom, UFMS, Brazil  
hora@facom.ufms.br

Romain Robbes  
PLEIAD Lab, DCC, University of Chile, Chile  
rrobbes@dcc.uchile.cl

Marco Tulio Valente  
ASERG Group  
DCC, UFMG, Brazil  
mтов@dcc.ufmg.br

Nicolas Anquetil  
RMod team, Inria Lille, France  
nicolas.anquetil@inria.fr

ABSTRACT

Commonly, software systems have public (and stable) interfaces, and internal (and possibly unstable) interfaces. Despite being discouraged, client developers often use internal interfaces, which may cause their systems to fail when they evolve. To overcome this problem, API producers may promote internal interfaces to public. In practice, however, API producers have no assistance to identify public interface candidates. In this paper, we study the transition from internal to public interfaces. We aim to help API producers to deliver a better product and API clients to benefit sooner from public interfaces. Our empirical investigation on five widely adopted Java systems present the following observations. First, we identified 195 promotions from 2,722 internal interfaces. Second, we found that promoted internal interfaces have more clients. Third, we predicted internal interface promotion with precision between 50%–80%, recall 26%–82%, and AUC 74%–85%. Finally, by applying our predictor on the last version of the analyzed systems, we automatically detected 382 public interface candidates.

CCS Concepts

- Software and its engineering → Software evolution; Maintaining software;

Keywords

Software Evolution; API Usage; Internal Interface Analysis

1. INTRODUCTION

Nowadays, developers implement their systems on top of frameworks and libraries [45], which are commonly used in software development to reuse functionalities [25] and increase productivity [5,36]. Often, these systems have both public and internal interfaces [5,10]. Public interfaces are stable, supported and documented (i.e., backward compatible), thus they can be safely used by client systems. In contrast, internal interfaces are unstable and unsupported (i.e., backward incompatible), intended to handle local functionalities, thus they should not be used by clients [3–5,10,12]. In the Java language, visibility modifiers may be used to prevent illegal references to internal elements. However, when this is not possible, developers may adopt naming conventions (guidelines) to clearly state that internal interfaces should not be used by clients [2]. In Eclipse, for example, internal interfaces include the word “internal”, and in the Java Development Kit (JDK), they start with “sun”.

Despite being discouraged, clients commonly use internal interfaces to access functionalities not supported by public ones [2–5,10,30], which may cause their systems to fail when these interfaces evolve [3–5]. In JDK, internal interfaces have been used by various applications for different reasons over the course of Java’s history [30], posing challenges when they change and decreasing client portability. In Eclipse, internal interfaces are often used by plugins: 44% of 512 analyzed plugins depended on internal interfaces and these clients usually had incompatibilities problems when Eclipse evolved [5]. To better motivate the study presented in this paper, we replicated this analysis at an ultra-large-scale level: with the support of the Boa2 infrastructure [13], we detected that 2,277 (23.5%) out of 9,702 Eclipse client projects, stored in GitHub, depended on internal interfaces. In fact, any change in these internal elements such as removal/renameing of attributes or methods, modification of method parameters, etc. may break client systems. For instance, the removal of the internal interface StatusDialog in Eclipse caused an important client (JBoss) to fail. As a solution to mitigate these risks and help client developers, API producers may promote some internal interfaces to public ones. For example, as documented in the commit in Figure 1, the internal interface BaseJavaElementContentProvider was promoted to the public StandardJavaElementContentProvider in Eclipse. In fact, Eclipse has more than 4K internal interfaces, but why exactly was BaseJavaElementContentProvider promoted? Unfortunately, in practice, API producers have no assistance to identify public interface candidates (i.e., internal interfaces that should be public). Thus,

1JDK/Oracle note: https://goo.gl/o7TN0P.
2http://boa.cs.iastate.edu
4Commit: http://goo.gl/DfZg7z.
the promotions occur at a slow pace, causing delay to client developers to benefit from stable and supported interfaces.

In this paper, we study the transition from internal to public interfaces. We aim to help both API producers and clients. API producers can deliver a better product by detecting public interface candidates while API clients can benefit sooner from stable and supported interfaces.

We initially investigate whether there is a relationship between internal interface usage and promotion to public. However, measuring external interface usage is costly and may not provide reliable observations if all clients are not identified. To avoid this problem, we decide to measure interface usage only from within the systems under analysis, namely domestic internal interface usage. Moreover, we investigate whether internal interface promotion can be detected and predicted. Therefore, we propose the following research questions:

**RQ1.** Is there a relationship between domestic internal interface usage and promotion?

**RQ2.** Can we predict that an internal interface will be promoted to a public one?

**RQ3.** Can we detect that an internal interface is a candidate to be promoted to a public one?

We analyze five real-world systems: Eclipse, JUnit, Hibernate, jBPM, and ElasticSearch. Our empirical investigation presents the following observations. First, from 2,722 analyzed internal interfaces, we found that 195 were promoted to public. Second, promoted internal interfaces are more domestically used than non-promoted ones. Third, we predict internal interface promotion with precision between 50%–80%, recall 26%–82%, and AUC 74%–85%. Finally, by applying our predictor on the analyzed systems, we automatically detected 382 public interface candidates; by analyzing their external usage at a large-scale level, we confirm they are relevant candidates. Thus, the contributions of this paper are summarized as follows:

1. We study the frequency of internal interfaces and their promotion to public in real-world systems.

2. We relate internal interface promotion with their usage to better understand this phenomenon.

3. We propose a technique to detect/predict public interface candidates to help both API producers and clients.

**Structure of the paper:** In Section 2, we discuss internal and public interfaces in more details. We describe our experiment design in Section 3. We present the experiment results/discussion in Section 4, and summary/findings in Section 5. We state the threats to validity in Section 6, and we present related work in Section 7. Finally, we conclude the paper in Section 8.

# 2. INTERNAL AND PUBLIC INTERFACES

In this section we define the concept of internal and public interfaces adopted in this study, present real-world examples of internal interface promotions, and discuss the challenges API producers face to identify public interface candidates.

## 2.1 Definition

Interface is an access point to a component that client systems can reference to reuse functionalities [12]. In many systems, it is common practice to adopt the notion of public and internal interfaces [3–5, 10, 12, 30]. Public interfaces are expected to be stable, supported, and documented. As they provide backward compatibility, their clients should not fail when these interfaces evolve. In contrast, internal interfaces refer to implementation functionalities that were not originally designed to be used by clients [3–5]. When clients use internal interfaces, they are subjected to unstable, unsupported, and undocumented services (i.e., backward incompatibility), consequently, they may fail when these internal interfaces evolve. Notice that Java access modifiers restrict access at the class, subclass, or package boundaries. However, when a large system spans several packages, there is no easy way to selectively grant access of certain types to a given set of packages. Thus, a common practice is to permit worldwide access to types that need to be accessed by other packages, but use naming conventions (such as internal packages) to discourage API clients to use them. For example, in Eclipse, internal interfaces are implemented in packages with the word “internal” (e.g., org.eclipse.jdt.internal.ui.JavaPlugin) while in the JDK, internal interfaces are the ones in packages with the prefix “sun” (e.g., sun.misc.Unsafe) [30].

We present below three internal interface guidelines extracted from documentation of Eclipse, jBPM, and JDK:

**Eclipse.** Packages containing only implementation details have “internal” in the package name. Legitimate client code must never reference the names of internal elements. Client code that oversteps the above rules might fail on different versions and patch levels of the platform.

**jBPM.** Expert users can still access internal classes but should be aware that they should know what they are doing and that internal API might still change in the future.

**JDK.** The sun.* packages are not part of the supported, public interface. A Java program that directly calls into sun.* packages is not guaranteed to work on all Java-compatible platforms. In fact, such a program is not guaranteed to work even in future versions on the same platform.

---

[^AUC]: AUC (area under the curve) is a measure for classifiers. AUC ≥ 70% is considered reasonably good [26, 43, 44].
2.2 Internal Interface Usage

In practice, the literature shows that client developers commonly use internal interfaces [2–5, 10, 30], despite being discouraged. For example, internal interfaces of Eclipse [4] and JDK [30] are often used by clients due to several reasons. As a result, they suffer from incompatibilities problems when these interfaces evolve [3–5, 10].

We replicated this analysis at an ultra-large-scale level with the support of the Boa infrastructure [13]. Boa is a software repository with over 7,8 million GitHub projects written in several programming languages. It includes 262K Java projects, which have over 125 million import statements. By mining these import statements, we detected 9,702 client projects depending on Eclipse. Finally, we found 2,277 (23.5%) of these client projects depending Eclipse internal interfaces. Thus, we show at an ultra-large-scale that internal interfaces are often used, reinforcing the results of previous studies.

2.3 Internal Interface Promotion

In order to handle the problems related with internal interface usage, API producers occasionally promote internal interfaces to public. We present next three real examples of internal interface promotion:

**MultipleFailureException in JUnit.** An API producer stated: “Create MultipleFailureException in org.junit.runners.model and deprecate org.junit.internal.runners.model. This allows client developers to properly handle multiple exceptions without depending on internal classes”.

**CharOperation in Eclipse.** An API client requested: “CharOperation is technically internal API to JDT-Core, but it seems generally useful enough to use elsewhere. I would like to see it made public API”. An API producer readily agreed with the suggestion: “We could indeed surface it into an API package”.

**NodeFinder in Eclipse.** An API client requested: “The NodeFinder class is part of the package org.eclipse.jdt.internal.corext.dom and provides very useful logic. Would be nice if NodeFinder becomes part of the AST public API”. In this case, the API producer did not accept the proposition at first, but it was accepted some years later.

In the first example, the internal interface was promoted by the API producer when he realized that API clients could benefit from this interface. In contrast, in the second and third examples, the suggestion came from API clients themselves: one was immediately accepted while the other was later accepted. In all the examples, it took years to promote the interfaces. For example, the JUnit interface MultipleFailureException was created in 2007, but only promoted to public three years later, in 2010. Similarly, the Eclipse issue asking the promotion of NodeFinder was opened in 2004 but only accepted four years later, in 2008. Thus, waiting for promotions may be inefficient, because it is time-consuming and client-dependent.

2.4 Problem: How to Identify Public Interface Candidates?

In practice, API producers have no assistance to identify public interface candidates (i.e., internal interfaces that should be public). Therefore, the promotions occur at a slow pace or only under specific requests. Based on that, one important question appears: is it possible to automatically identify public interface candidates? Answering this question brings two practical results. First, API producers can deliver a better product, which is less likely to break client systems. Second, API clients can benefit sooner from more stable and supported interfaces. In the next sections we aim to answer this question.

3. EXPERIMENT DESIGN

3.1 Selecting the Case Studies

For this study, we select five real-world systems: Eclipse JDT (Eclipse’s Java development tools), JUnit (testing framework), Hibernate (library with Object/Relational mapping support), jBPM (business process management suite), and ElasticSearch (distributed RESTful search engine). They adopt the convention of internal interfaces by using “internal” in their packages. Table 1 presents an overview of these systems in number of stars, commits, and releases.

<table>
<thead>
<tr>
<th>System</th>
<th>Stars</th>
<th>Commits</th>
<th>Releases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>77</td>
<td>57,665</td>
<td>7,764</td>
</tr>
<tr>
<td>JUnit</td>
<td>3,844</td>
<td>2,062</td>
<td>20</td>
</tr>
<tr>
<td>Hibernate</td>
<td>1,346</td>
<td>6,049</td>
<td>95</td>
</tr>
<tr>
<td>jBPM</td>
<td>391</td>
<td>2,685</td>
<td>55</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>12,472</td>
<td>14,281</td>
<td>127</td>
</tr>
</tbody>
</table>

3.2 Extracting Internal and Public Interfaces

We then extract internal and public interfaces from the selected case studies. For each system, we extract the interfaces from the import statements (e.g., Eclipse interfaces start with org.eclipse, JUnit interfaces start with org.junit or junit, etc.). If an interface includes the word “internal” in its name, it is tagged as an internal one, otherwise it is tagged as a public one. This process takes into account the full source code history of the system under analysis. Therefore, if an interface was referenced in the past, but it is not referenced anymore, it is still considered. This decision was made because we intend to search internal interface promotions, and this is possible by taking into account code history (cf., subsection 3.3). Moreover, we want to assess internal interface usage over time, not only in the current version of a system (cf., subsection 3.4).

Table 2 presents the number of public and internal interfaces. Eclipse presents the highest proportion of internal interfaces: 51% (4,580 out of 8,921). In contrast, ElasticSearch presents the lowest proportion: 2% (155 out of 7,214). Considering all systems, 21% of the interfaces are internal (6,085 out of 28,503). The results presented in Table 2 confirm that these systems adopt internal interfaces in their design, thus they are relevant for our analysis.

---

9http://boa.cs.iastate.edu/stats/index.php
10Commit: http://goo.gl/AWCTu
13We analyze the subprojects Core, UI, and Debug.
3.3 Searching Internal Interface Promotions

We search internal interface promotions by analyzing the source code history of the case studies. Our heuristic to search internal interface promotions is detailed below. For each system, we mine every file change on its code history. A promotion from interface Internal to Public is detected when two constraints are satisfied:

1. There is at least a file change that removes only one reference to Internal and adds only one reference to Public, and
2. The class names of the references remain the same or have an suffix/prefix added/removed.

Thus, a promotion from interface Internal to Public needs (1) to happen in a file change in which it is the only changing reference, and (2) to have the same or similar class names. The first constraint increases the confidence on the promotion while the second helps to filter out noisy promotions.

Table 3 shows the internal interfaces that can be verified as promoted to public or not. Eclipse has the highest absolute value: 145 promotions. jBPM presents the highest proportion of promoted interfaces: 34% (17 out of 50). Considering all systems, 7% of the internal interfaces are promoted to public (195 out of 2,722). In 115 promotions the class name remained the same, while in 5 cases the promotion but incorrect). For example, the transition from org.elasticsearch.client.Client to org.elasticsearch.client.internal.Client is also correct; notice that the class name changed from InternalClient to Client, as allowed by clause 2 of the heuristic.

Second, we also inspected 50 randomly selected transitions not classified as promotions by our heuristic to find false negatives (i.e., not classified as promotion but correct). For example, the transition from org.eclipse.jdt.core.compiler.util.CharOperation to org.eclipse.jdt.core.compiler.util.Strings is marked as incorrect after looking at code examples, because the services provided by the internal class Strings are only partially replaced by the ones in the public CharOperation.

With this manual analysis, we detected 92% of true positives (46 out of 50) and 12% of false negatives (6 out of 50). Thus, the risk of incorrect or missing classification is low.

3.4 Measuring Internal Interface Usage

As concluded in the previous section, internal interfaces are frequently used by external clients [3–5,10]. In this work, we study whether there is a relationship between interface usage and promotion to public. However, one important question appears: how can we assess whether internal interfaces are used by external clients? Answering this question requires two steps. First, we should select a large set of clients and, second, process each external client looking for internal interface usage. In fact, this is a costly process and may be inefficient if a representative sample of clients is not correctly identified. Therefore, to overcome this challenge, we decide to measure internal interface usage from within the system under analysis, namely domestic internal interface usage.

Domestic vs. External Usage: Next, we assess whether domestic internal interface usage can be confidently considered in our study. Our goal is to verify whether domestic usage of internal interfaces has a relation with external usage. If we find that the most domestically used internal interfaces are likely to be externally used, we can confidently say that domestic usage is a good proxy for external usage. For that analysis, we extracted external clients from the Boa ultra-large-scale software repository [13]. We consider that an external client project uses an internal interface when it contains at least one reference to it. The domestic clients are extracted from the system under analysis itself, and they are represented as the client classes. To perform the analysis, (i) we sorted the internal interfaces by the number of domes-

http://goo.gl/KXC5SR

Table 2: Number of public and internal interfaces.

<table>
<thead>
<tr>
<th>System</th>
<th>Interfaces</th>
<th>Type</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Public</td>
<td>Internal</td>
</tr>
<tr>
<td>Eclipse</td>
<td>8,921</td>
<td>4,341</td>
<td>4,580 (51%)</td>
</tr>
<tr>
<td>JUnit</td>
<td>944</td>
<td>803</td>
<td>141 (15%)</td>
</tr>
<tr>
<td>Hibernate</td>
<td>6,821</td>
<td>5,875</td>
<td>946 (14%)</td>
</tr>
<tr>
<td>jBPM</td>
<td>4,603</td>
<td>4,340</td>
<td>263 (6%)</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>7,214</td>
<td>7,059</td>
<td>155 (2%)</td>
</tr>
<tr>
<td>Total</td>
<td>28,503</td>
<td>22,418</td>
<td>6,085 (21%)</td>
</tr>
</tbody>
</table>

Table 3: Number of promoted and non-promoted internal interfaces in the train and test dataset.

<table>
<thead>
<tr>
<th>System</th>
<th>Internal Interfaces</th>
<th>Promotion</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Eclipse</td>
<td>2,155</td>
<td>2,010</td>
</tr>
<tr>
<td>JUnit</td>
<td>92</td>
<td>83</td>
</tr>
<tr>
<td>Hibernate</td>
<td>328</td>
<td>315</td>
</tr>
<tr>
<td>jBPM</td>
<td>50</td>
<td>33</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>97</td>
<td>86</td>
</tr>
<tr>
<td>Total</td>
<td>2,722</td>
<td>2,527</td>
</tr>
</tbody>
</table>

\[\text{(i) we sorted the internal interfaces by the number of domestic usage as promoted to public or not.} \]

\[\text{(ii) we also inspected 50 randomly selected transitions not classified as promotions by our heuristic to find false negatives.} \]

\[\text{Table 3 shows the 2,722 (out of 6,085) labeled internal interfaces (see the train and test dataset in Table 6).} \]

\[\text{Similarly, the transition from org.elasticsearch.client.internal.InternalClient to org.elasticsearch.client.internal.Client} \]

\[\text{16Ex: https://goo.gl/w6075d and https://goo.gl/M4upG8} \]

\[\text{17Results are available at: https://goo.gl/aqMars} \]
tic clients, and (ii) we computed the percentage of internal interfaces being used by external clients in the top-10% and top-n from this sorted list, where n is the number of internal interfaces in a system. Table 4 presents the results. In JUnit, for example, 44% of the top-10% most domestically used internal interfaces have external clients; this percentage is 22% for the top-n. As seen in Table 4, in all the cases, the top-10% are much more likely to have external clients than the interfaces in the top-n in general. Therefore, this analysis confirms that domestic usage can be in fact used as a proxy for external usage. Further, Dagenais and Robillard successfully used internal usage of evolving interfaces to recommend evolution rules to external clients [10], thus showing that internal and external usage patterns of interfaces have a degree of similarity.

Table 4: External usage of most domestically used internal interfaces.

<table>
<thead>
<tr>
<th>System</th>
<th>% of external usage in top-10%</th>
<th>% of external usage in top-n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>JUnit</td>
<td>44</td>
<td>22</td>
</tr>
<tr>
<td>Hibernate</td>
<td>75</td>
<td>59</td>
</tr>
<tr>
<td>jBPM</td>
<td>100</td>
<td>44</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>80</td>
<td>43</td>
</tr>
</tbody>
</table>

Selected Domestic Usage Metrics: Table 5 presents five metrics related to domestic internal interface usage adopted in this study, which are computed per system. The metrics packages and classes measure the number of distinct containers referencing an internal interface. As we analyze the source code history, we also measure the number of distinct commits adding references to an internal interface (metric commits) and the number of distinct developers authoring these commits (metric developers). Finally, the metric time measures the number of days between the first and the last usage of an internal interface; this metric is intended to verify whether an internal interface is still attracting new (domestic) clients over time.

Table 5: Domestic internal interface usage metrics.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#packages</td>
<td>Number of distinct packages referencing an internal interface</td>
</tr>
<tr>
<td>#classes</td>
<td>Number of distinct classes referencing an internal interface</td>
</tr>
<tr>
<td>#commits</td>
<td>Number of distinct commits that added references to an internal interface</td>
</tr>
<tr>
<td>#developers</td>
<td>Number of distinct developers that added references to an internal interface</td>
</tr>
<tr>
<td>#time</td>
<td>Number of days between the first and the last usage of an internal interface</td>
</tr>
</tbody>
</table>

3.5 Classifying Internal Interfaces

Based on the usage metrics proposed in the previous subsection, we investigate whether internal interface promotion can be predicted (by analyzing past code) and detected (by analyzing current code). In order to support this analysis, we classify an internal interface as absent or present in the current source code, and as promoted or non-promoted.

If internal interfaces are absent or promoted (cases A, B, and C in Table 6), they had the opportunity to be promoted in the past. In other words, these absent or promoted internal interfaces can be used to train and test the proposed approach, i.e., they are a labeled dataset. In contrast, if internal interfaces are present and non-promoted (case D), they are still candidates to be promoted (because they are present in current version). That is to say, these present and non-promoted internal interfaces can only be used to test the proposed approach, i.e., they are an unlabeled dataset.

We take into account the train and test dataset (cases A, B, and C) to answer RQ2 about promotion prediction. These cases involve 2,722 labeled internal interfaces, as presented in Table 3. Finally, we consider the test dataset (case D) to answer RQ3 about detection of public interface candidates, which involve 3,363 unlabeled internal interfaces (totalizing the 6,085 shown in Table 2).

Table 6: Internal interface classification.

<table>
<thead>
<tr>
<th></th>
<th>Non-promoted</th>
<th>Promoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absent</td>
<td>Train &amp; test (A)</td>
<td>Train &amp; test (B)</td>
</tr>
<tr>
<td>Present</td>
<td>Test (D)</td>
<td>Train &amp; test (C)</td>
</tr>
</tbody>
</table>

4. Results

RQ1. Is there a relationship between domestic internal interface usage and promotion?

In this research question, we aim to understand factors of promoted internal interfaces. We hypothesized that promoted internal interfaces are more likely to have clients.

Approach. We compare the values of the five domestic usage metrics (i.e., packages, classes, commits and developers and time) between promoted and non-promoted internal interfaces. We first compare the distribution between the two groups with box plots. Then, we analyze the statistical significance of the difference between the two groups of internal interfaces by applying the Mann-Whitney U test at p-value = 0.05. The null hypothesis is that promoted and non-promoted internal interfaces present similar distribution. To show the effect size of the difference between the two groups, we compute Cliff’s Delta (d), which is an effect size measure. 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, as in other studies [14, 44].

Results. First, we characterize (promoted and non-promoted) internal interfaces with respect to their usage by domestic clients. Clients are defined in terms of distinct packages, classes, commits, and developers using the internal interface. Figure 2 presents the distribution of clients for the systems under analysis. Each box plot shows on the left the distribution of promoted internal interfaces and on the right of non-promoted internal interfaces. Overall, we clearly see the higher distribution of promoted internal interfaces in all the analyzed usage metrics. In Eclipse, for example, the median for packages is 4 against 1, for classes 8/2, for commits 4/1,
and p-values for usage metrics with p-value < 0.05. Effect-size is large (i.e., d-value > 0.474) for all the systems, and p-value < 0.001 is most of the cases.

Table 7: Comparison of promoted and non-promoted internal interface usage. Effect-size is large for all the metrics (d-value > 0.474). ***: p-value < 0.001, **: p-value < 0.01, and *: p-value < 0.05.

<table>
<thead>
<tr>
<th>System</th>
<th>Pkg</th>
<th>Cls</th>
<th>Com</th>
<th>Dev</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>1.4***</td>
<td>1.2***</td>
<td>1.4***</td>
<td>1.3***</td>
<td>1.1***</td>
</tr>
<tr>
<td>JUnit</td>
<td>2.3***</td>
<td>2.2**</td>
<td>2.4**</td>
<td>1.2*</td>
<td>-</td>
</tr>
<tr>
<td>Hibernate</td>
<td>1.0***</td>
<td>0.9**</td>
<td>0.9**</td>
<td>1.4***</td>
<td>0.6**</td>
</tr>
<tr>
<td>jBPM</td>
<td>1.5***</td>
<td>1.2***</td>
<td>1.1**</td>
<td>0.88*</td>
<td>-</td>
</tr>
<tr>
<td>ElasticS.</td>
<td>2.3***</td>
<td>3.7***</td>
<td>1.6**</td>
<td>1.9**</td>
<td>2.1**</td>
</tr>
</tbody>
</table>

In summary, promoted internal interfaces are statistically significantly different from internal interfaces that are not promoted in most of the usage metrics. Overall, internal interfaces that are promoted are used by more domestic packages, classes, commits, and developers, and they attract newer clients over time.

RQ2. Can we predict that an internal interface will be promoted to a public one?

Based on the domestic usage of internal interfaces, we investigate whether internal interface promotion can be predicted by analyzing past source code.

Approach. We compare the importance of multiple metrics on internal interface promotion. We build a random-forest classifier to predict whether an internal interface will be promoted, given the values of the metrics. We choose the random-forest classifier because it is known to have several advantages, such as being robust to noise and outliers [44]. In addition, the classification power of random-forest classifiers has been demonstrated by its application to automate many software engineering tasks [1, 11, 26, 32, 38, 44], many of those with unbalanced data. We use 10-fold cross validation to evaluate the effectiveness of our model. We train and test the classifier with internal interfaces that are absent or promoted (cases A, B, and C in Table 6).

Evaluation. We assess the effectiveness of the classifier in correctly predicting promoted internal interfaces. We use precision, recall, F-measure and AUC (area under curve) to measure its effectiveness, which are commonly adopted in classification tasks [11, 24, 44]. Precision and recall measure the correctness and completeness, respectively, of the classifier in predicting whether an internal interface is promoted. F-measure is the harmonic mean of precision and recall. AUC is the area under the Receiver Operating Characteristic (ROC) curve. AUC is robust toward unbalanced data [39]. AUC ≥ 70% is considered reasonably good [26, 43, 44].

Results. Table 8 shows the prediction results when considering the domestic metrics. We predict internal interface promotion with precision between 50%–80%, recall 26%–82%, F-measure 35%–60%, and AUC 74%–85%. Hibernate presents the best precision (80%) and jBPM the worst (50%). jBPM presents the best recall (82%) and Eclipse
Promotion Examples. When an internal interface is promoted to public, it means that it is generic enough to be reused by clients. API producers may detect internal interfaces that should be promoted, such as the promotion of MultipleFailureException in JUnit, or API clients may demand the promotion, such as the promotions of CharOperation and NodeFinder in Eclipse (c.f. subsection 2.3).

Internal interfaces may be deliberately promoted to avoid bugs in clients. As presented in Table 9, the Eclipse internal interfaceorg.eclipse.jdt.internal.ui.dialogs.StatusDialog was removed, causing an important client to fail (JBoss). The API client requested the internal interface to be added back. However, the API producer noted that the removed interface was internal, thus it was unstable, and could be removed. After all, the internal interface was promoted to public to better support client systems, helping them to be bug-free. Interfaces are also promoted in cases they are mistakenly projected as internal, as presented in the JUnit example in Table 9. In this case, extensions could not be implemented by clients without necessarily depending on the internal interfaceorg.junit.internal.AssumptionViolatedException. After a client request, the promotion was confirmed in the release notes of JUnit.

RQ3. Can we detect that an internal interface is a candidate to be promoted to a public one?

In this final research question we aim to detect public interface candidates in the current source code of the analyzed case studies. The idea is to help API producers and clients by pointing possible internal interfaces (still present in source code) that are candidates to public.

Approach. In order to detect candidates, we train and test a random-forest classifier with the five usage metrics adopted in the previous research question. We train the classifier with internal interfaces that are in the train and test dataset (Table 6). Finally, to detect candidates, we test the classifier with internal interfaces that are in the test dataset (Table 6). In the testing step, we label an internal

TABLE 8: Prediction results (percentages).

<table>
<thead>
<tr>
<th>System</th>
<th>Prec</th>
<th>Rec</th>
<th>F-m</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>54</td>
<td>26</td>
<td>35</td>
<td>75</td>
</tr>
<tr>
<td>JUnit</td>
<td>75</td>
<td>33</td>
<td>46</td>
<td>76</td>
</tr>
<tr>
<td>Hibernate</td>
<td>80</td>
<td>31</td>
<td>44</td>
<td>84</td>
</tr>
<tr>
<td>jBPM</td>
<td>50</td>
<td>82</td>
<td>62</td>
<td>74</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>66</td>
<td>72</td>
<td>69</td>
<td>85</td>
</tr>
</tbody>
</table>

interface as a candidate to public when the classifier reports a promotion probability ≥ 50%.

RESULTS. Table 10 reports the number of public interface candidates in each system (i.e., interfaces that satisfy the 50% threshold). Eclipse presents the highest number of public interface candidates: 298. Next, jBPM has 53 candidates, ElasticSearch has 17, and Hibernate has 10. JUnit presents only 4 candidates. In summary, we automatically detect 382 public interface candidates in the current version of the analyzed systems. These internal interfaces have domestic usage metrics similar to past internal interfaces that were already promoted.

TABLE 9: Examples of internal interface promotions.

<table>
<thead>
<tr>
<th>System</th>
<th>Promotion &amp; Explanation</th>
</tr>
</thead>
</table>
| Eclipse     | An API client requested:  
“Since AssumptionViolatedException is internal API, you cannot write a rule or runner without crossing the boundary into internal API”. JUnit release notes show that the request was accepted: “In JUnit 4.11 and earlier, if you wanted to write a custom runner that handled AssumptionViolatedException, you needed to import an internal class. Now you can import the public org.junit.AssumptionViolatedException”.

| JUnit       | An API client requested:  
“Since AssumptionViolatedException is internal API, you cannot write a rule or runner without crossing the boundary into internal API”. JUnit release notes show that the request was accepted: “In JUnit 4.11 and earlier, if you wanted to write a custom runner that handled AssumptionViolatedException, you needed to import an internal class. Now you can import the public org.junit.AssumptionViolatedException”.

<table>
<thead>
<tr>
<th>System</th>
<th>Internal Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>2,425</td>
</tr>
<tr>
<td>JUnit</td>
<td>49</td>
</tr>
<tr>
<td>Hibernate</td>
<td>618</td>
</tr>
<tr>
<td>jBPM</td>
<td>213</td>
</tr>
<tr>
<td>ElasticSearch</td>
<td>58</td>
</tr>
<tr>
<td>Total</td>
<td>3,363</td>
</tr>
</tbody>
</table>

External Usage of Candidates. To assess the quality of public interface candidates, we verify whether they are used by external clients. If they are used by many clients, this

The best F-measure is found in ElasticSearch (69%) and the worst in Eclipse (35%). Finally, considering AUC, ElasticSearch presents the best results (85%) and jBPM the worst (74%). Notice that AUC beyond 70% is considered reasonably good. Additionally, in the Software Engineering domain, many proposed recommenders have AUC values between 70%–80% [26, 43, 44]. These results confirm the impact of the analyzed metrics on predicting internal interface promotion.
may be an indication that these interfaces are in fact relevant public candidates. Otherwise, this may indicate that they are not relevant public candidates. To do so, we compare external usage of three internal interface groups: (i) public interface candidates (i.e., internal interfaces with more probability to be promoted), (ii) non-candidates (i.e., internal interfaces with less probability to be promoted), and (iii) randomly selected internal interfaces (which may include both candidates and non-candidates). Finally, to collect external clients, we rely on the ultra-large-scale Boa software repository [13]. We then verify whether these clients are depending on the three groups of internal interfaces.

In a first analysis, we consider all case studies, thus, each group is formed by 382 internal interfaces (see Table 10). Figure 3 presents each external usage distribution. For the candidates, the first, second, and third quartiles are 4, 11 and 23: 50% of the public interface candidates have 11 client projects. In contrast, in the other two groups external usage is much smaller. For the random group, the first, second, and third quartiles are 1, 2 and 6. For the non-candidates, the first, second, and third quartiles are 0, 1, and 5. By applying the Mann-Whitney test, we verify that the difference between candidates and the other two groups are statistically significant ($p$-value < 0.001).

![Figure 3: External usage of candidates and non-candidates considering all case studies.](https://goo.gl/Um3f0d)

By replicating previous studies on internal interface usage, we show at an ultra-large-scale level that internal interfaces are often used (we detected 2,277 out of 9,702).

5. SUMMARY AND FINDINGS

Clients often use internal interfaces. We show that internal interfaces are often used by clients: 2,277 (23.5%) out of 9,702 Eclipse client projects depended on internal interfaces. This result complements the findings of previous studies on internal interfaces usage [2–5, 10, 30]. Based on these results, we conclude:

- Internal interfaces are not frozen. They may be promoted to public when API producers or clients discover that they can be reused (for example, we found 195 promotion in our study).

Promoted and non-promoted internal interfaces present distinct usage patterns. RQ1 shows that internal interfaces that are promoted are statistically different from internal interfaces that are not promoted. We found that promoted internal interfaces are domestically used by more packages, classes, commits and developers, and that they tend to attract newer clients over time. Therefore, we conclude the following:

- Distinct domestic usage pattern is found in promoted and non-promoted internal interfaces. Based on such difference, recommenders can be built to detect public interface candidates. To support the detection, recommenders can confidently rely on domestic usage of internal interfaces.

---

Footnotes:

20 https://goo.gl/Um3f0d

21 Eclipse public interface candidates: https://goo.gl/dOhFJi
**Internal interface promotions can be predicted with high confidence.** Based on the domestic usage metrics of the systems under analysis, we predict internal interface promotion with precision between 50%-80%, recall 26%-82%, and AUC 74%-85%, as presented in RQ2. Based on that, we conclude:

| By using usage metrics computed at domestic level, classifiers with good performance (i.e., AUC ≥ 70%) can be produced to predict internal interface promotion. |

**Public interface candidates can be detected in current source version.** By running a random-forest classifier in the current source code of the systems under analysis (RQ3), we detect 382 public interface candidates, i.e., internal interfaces that should be public. Eclipse presents the highest number of public interface candidates (298), followed by jBPM (53) and Elasticsearch (17). Thus, we conclude:

| In order to help both API producers and clients, we are able to automatically detect public interface candidates (we found 382 in our case studies). |

**Public interface candidates are externally used.** By relying on domestic usage of internal interfaces, we discovered public interface candidates. Those candidates are more likely to have external clients (11, on the median) than non-candidates (1 client, on the median). Thus, we conclude:

| Domestic usage of internal interfaces can be used as good estimator of their external usage. This suggests that detected internal interface candidates are in fact relevant. |

### 6. THREATS TO VALIDITY

**Construct Validity.** The construct validity is related to whether our study reflects real-world situations.

**Internal Interface Guideline Adoption.** One threat may be the possibility that the usage of internal interfaces does not happen in other systems than the ones analyzed in our study. However, by mining popular Java systems in GitHub, we detected several adopting guidelines of internal interfaces: 34 out of 350 (10%). Thus, this reinforces that the use of internal interfaces is common practice in relevant Java systems, making our approach also suited for them. Further, since the issue stems from a lack of granular access modifiers between Java packages, the problem is widespread, and thus other systems may use different guidelines to the same effect. The approach could be adapted to these specific cases.

**Internal Interface Promotion.** Another possible threat is related to the correctness and completeness of the detected internal interface promotions. As presented in subsection 3.3, we paid special attention when searching for internal interface promotions. This process involved (i) the definition of a heuristic to detect promotions, and (ii) the manual analysis of 100 promotions with the support of code examples and commit logs in order to assess false positives and negatives. We found that incorrect or missing classification is low, reducing the risk of this threat.

**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 and Random-forest classifier) when reporting our results in RQs 1-3. In RQ3, the 50% threshold was set to ensure reasonable precision and recall on the detected candidates; further study may vary this threshold to evaluate its effect on candidate detection. Moreover, even though our classifier reported good performance, we validated the public interface candidates in the context of external usage. An alternative validation for RQ3 is with the help of core/expert developers on the systems under analysis, which remains future work.

**Association and Causation.** The purpose of our study is to examine whether there are factors (i.e., the usage metrics) that are associated to internal interface promotion. Notice, however, that association does not imply causation [8, 9]. Therefore, more advanced statistical analysis, for example, causal analysis [40], can be adopted to further extend our study.

**External Validity.** The external validity is related to the possibility to generalize our results.

- We focused on the analysis of widely adopted, large-scale and real-world Java systems, therefore they are credible and representative case studies. These systems are stored in GitHub, the most popular code repository nowadays, thus they their source code is easily accessible. Despite these observations, our findings — as usual in empirical software engineering — cannot be directly generalized to other systems, specifically to systems implemented in other programming languages. Therefore, our study should be carried out on other systems, possibly written in other languages.

### 7. RELATED WORK

**7.1 Internal Interface Usage**

Businge et al. [3] study the survival of Eclipse plugins, and classify them in two categories: the ones depending on internal interfaces (also known as bad or non-APIs) and the ones depending only on public interfaces (good or APIs). They verify that despite being discouraged, client developers often use internal interfaces. In an extension study [5], the authors show that 44% of 512 analyzed Eclipse plugins depend on internal interfaces, which are a source of incompatibilities problems when Eclipse evolves. In a related study [4], the same authors investigate the reasons why developers use internal interfaces. They detect cases where developers do not read documentation/guidelines, and where they deliberately use internal interfaces to benefit from advanced functionalities. Mastrangelo et al. [30] show that client projects frequently use internal interfaces provided by JDK. Even though it is unsafe, the authors found several usage patterns of the internal interface sun.misc.Unsafe, such as to allocate objects without invoking constructors and to load classes without security checks. In this context, Boulanger and Robillard [2] propose a tool that restricts (or permits) access to the implementation of a service. Vidal et al. [46] study information hiding at a large-scale level in order to better understand what should constitute a public interface.

In summary, the literature shows that the usage of internal interface occurs in practice, causing real problems to clients. We complement these studies by showing that internal interfaces may be promoted to public, and we propose a machine learning technique that alleviates these problems by identifying public interface candidates.
7.2 API Migration/Evolution

Many approaches have been developed to support API evolution and reduce the efforts of client developers when facing API migration. Chow and Notkin [6] present an approach where API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [15] propose CatchUp, a tool that uses an IDE to capture and replay refactorings related to API evolution. Other studies are intended to mine API evolution rules from source code history. Kim et al. [23] automatically infer rules from structural changes in source code.Hora et al. [19,21] propose tools to keep track of API evolution by mining fine-grained code changes. Kim et al. [22] introduce LSDiff to support computing differences between two system versions. Nguyen et al. [37] present LibSync, which uses graph-based techniques to help developers migrate from one framework version to another. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting systems to API evolution. Mileva et al. [35] also mine two version of a system to detect evolution of object usage, and help developers to ensure changes are systematically applied in source code.

Schafer et al. [42] mine framework usage change rules from client systems. Dagenais and Robillard [10] introduce SemDiff, which suggests replacements for framework elements based on how they adapt to their own changes. Meng et al. [33] propose a history-based matching approach (HiMa) to support framework evolution. Wu et al. [47] present an approach that combines call dependency and text similarity analyses to produce evolution rules. 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. [16–18] focus on the extraction of domain-specific API evolution rules.

API evolution is also studied in the context of software ecosystems. McDonnell et al. [31] investigate API stability and adoption on a small-scale Android ecosystem. They have found that Android APIs are evolving fast while client adoption is not catching up with the pace of API evolution. In a large-scale study, Robbes et al. [41] investigate the impact of API deprecation in a software ecosystem. Hora et al. [20] complement the previous study by analyzing the impact of API deprecation (not related to deprecation) at large-scale level. Both studies agree that software ecosystems do not react to API evolution due to reasons such as lack of deprecation messages and unaware developers.

Livshits and Zimmermann [27] propose to discover usage patterns over code history, such as method pairs (e.g., lock() must happen with unlock()). Some studies mine execution traces [28,29]. They extract rules via dynamic analysis to produce temporal rules (e.g., every call to m1() must be preceded by a call to m2()). Other studies focus on how the usage of APIs by client systems (i.e., API popularity) can contribute to better support software development [21,34].

Dig and Johnson [12] help developers to understand the requirements for migration tools, finding that 80% of the changes that break clients are refactorings. Cossette et al. [7] find that API incompatibility is hard to handle, concluding that API migration remains a challenging proposition.

In summary, related studies are intended to better understand API evolution and to propose solutions to API migration. None of them, however, study API evolution/migration in the context of internal interfaces.

8. CONCLUSION

To the best of our knowledge, this work is the first (i) to study internal interface promotion to better understand this phenomenon and (ii) to provide a technique to detect public interface candidates to help both API producers and clients. The study was done in the context of five real-world Java systems (Eclipse, JUnit, Hibernate, jBPM, and Elastic-Search). Three research questions were investigated to characterize, detect, and predict internal interfaces that should be promoted to public. We reiterate the most interesting conclusions from our experiment results:

1. Clients often use internal interfaces. At an ultra-large-scale level, we show that 2,277 (23.5%) out of 9,702 Eclipse client projects depended on internal interfaces.
2. Internal interface promotions happen in real-word systems. 7% (195 out of 2,722) of the internals are promoted to public, which may have high impact on clients.
3. Promoted and non-promoted internal interfaces present distinct domestic usage. Recommenders can be built to detect public interface candidates and confidently rely on domestic usage of internal interfaces.
4. Internal interface promotions can be predicted with high confidence. By adopting the domestic usage metrics, classifiers with good performance (AUC ≥ 70%) can be produced to predict internal interface promotion.
5. Public interface candidates can be detected in current source code. To help both API producers and clients, we are able to automatically detect public interface candidates (we found 382 in our case studies).
6. Public interface candidates are externally used and unstable. Domestic usage of internal interfaces is a good estimator of their external usage, reinforcing that the detected candidates are relevant.

Besides the results presented here, our findings point out that the access limitations enforced by the access modifiers implemented in the Java language may not be granular enough for large-scale systems in which several packages may need additional visibility, without granting unrestricted access to the rest of the world.

As future work, we plan to extend this research to other systems and programming languages. We also plan to further validate our detection approach with more client systems and with the help of core/expert developers. Another extension of our study is to decrease the number of false negatives of the proposed heuristic by employing techniques of interface name and body similarity in order to detect the missing promotions. Finally, based on our approach, we plan to implement a tool that can be easily used by API producers and clients when looking for public interface candidates.

9. ACKNOWLEDGMENTS

This research is supported by CNPq, FAPEMIG, and Fundec-MT (007/ 2015).

10. REFERENCES


