Computer Languages, Systems & Structures 46 (2016) 1–19

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Over-exposed classes in Java: An empirical study

S. Vidal [a,c,*], A. Bergel [b], J.A. Díaz-Pace [a,c], C. Marcos [a,d]

[a] ISISTAN, Tandil, Argentina
[b] Department of Computer Science (DCC), University of Chile, Chile
[c] CONICET, Argentina
[d] CIC, Argentina

ARTICLE INFO

Article history:
Received 4 December 2015
Received in revised form
29 March 2016
Accepted 27 April 2016
Available online 6 May 2016

Keywords:
Class accessibility
Modularity
Over-exposed classes
Java systems

ABSTRACT

Java access modifiers regulate interactions among software components. In particular, class modifiers specify which classes from a component are publicly exposed and therefore belong to the component public interface. Restricting the accessibility as specified by a programmer is key to ensure a proper software modularity. It has been said that failing to do so is likely to produce maintenance problems, poor system quality, and architecture decay. However, how developers uses class access modifiers or how inadequate access modifiers affect software systems has not been investigated yet in the literature.

In this work, we empirically analyze the use of class access modifiers across a collection of 15 Java libraries and 15 applications, totaling over 3.6M lines of code. We have found that an average of 25% of classes are over-exposed, i.e., classes defined with an accessibility that is broader than necessary. A number of code patterns involving over-exposed classes have been formalized, characterizing programmers' habits. Furthermore, we propose an Eclipse plugin to make component public interfaces match with the programmer's intent.

1. Introduction

The need to support modularity in software systems can be traced back to Parnas [1], and encapsulation has been emphasized by Scott [2] in his seminal book as follows:

> *"Encapsulation allows the implementation details of an abstraction to be hidden behind a simple interface."*

Many constructs offered by programming languages are designed to manage the accessibility of routines/functions, which guide code reuse and extensions. The Java language offers rich linguistic constructs to define accessibility of classes. The accessibility of a class is specified via the absence or the presence of a particular keyword (`public`, `protected`, `private`). Class access modifiers play an important role in achieving modularity as they regulate inter-module interactions. Despite its relevance in promoting modularity, the use of accessibility modifiers in Java programs has received scant attention in the literature. Understanding how developers assign a particular accessibility to their classes is relevant to address several design aspects that are sensitive to modularity. First, class accessibility reflects assumptions and decisions made by the

* Corresponding author.
E-mail addresses: svidal@exa.unicen.edu.ar (S. Vidal), abergel@dcc.uchile.cl (A. Bergel), adiaz@exa.unicen.edu.ar (J.A. Díaz-Pace), cmarcos@exa.unicen.edu.ar (C. Marcos).

http://dx.doi.org/10.1016/j.cl.2016.04.001
1477-8424/

developer: a public class is part of the public interface of its containing package and might be considered as an entry point by other developers who wish to use the package. For example, public classes are listed in automatically generated Javadoc documentation and suggested by code completion tools in programming environments. Second, programming environments (IDEs), such as Eclipse, Netbeans, and IntelliJ assist programmers when creating classes in their projects. Relying on programming environments is highly important for developers in terms of productivity and quality code. However, current IDEs sometimes make suboptimal "default" decisions about class accessibility that programmers might be unaware of, which can have a negative impact on the overall system modularity. Although method accessibility has been the topic of a number of works [3–5], to the best of our knowledge the usage of class access modifiers has not been yet investigated. Moreover, this article is a continuation (and an improvement) of a prior study on method accessibility [4].

We have conducted an extensive study on the use of class access modifiers. This article reports our findings from a collection of 30 Java applications (15 libraries and 15 plain applications). We have identified a code anomaly called *class over-exposure* that applies to classes. This anomaly refers to a class that is unnecessarily accessible (to other classes), therefore we qualify it as being over-exposed. Over-exposed classes might have their accessibility reduced without affecting the program behavior while increasing the modularity of the system. For example, if a class is accessed from other packages than its own, the class has to be public. In contrast, if a class *C* is *solely* referenced from classes defined in the same package of *C*, then *C* should *likely* be private to the package. We investigated different aspects of this phenomenon, namely: ratio of public classes in different types of applications, ratio of over-exposed classes, and evolution of access modifiers of classes across several program versions, among others. Naturally, it might happen that a class is voluntary exposed to meet future client requirements. We carefully consider this situation in our analysis.

The article addresses the following research questions:

- *RQ1 – Do plain applications declare less public classes than libraries / frameworks*? If plain applications exhibit a different profile than libraries and frameworks, then they should be treated differently in the analysis of over-exposure.
- *RQ2 – How many classes are actually over-exposed?* Understanding to what extent this code anomaly is present in source code is important to gauge the severity of the anomaly.
- *RQ3 – Do over-exposed classes address future client requirements?* A premise to have a public class, with no actual public usage (yet), is that the class should be used by third-party components in the future. Answering this question can shed light on whether this expectation is fulfilled.
- *RQ4 – Are classes over-exposed since their first implementation, or do they become so over time?* This research question is important to figure out how over-exposed classes evolve over time.
- *RQ5 – Do over-exposed classes negatively impact the general software health?* Determining whether there is a perceptible degradation of the overall source code quality is relevant to formulate an adequate response to this phenomena.

Answering the questions above provides a great insight on how developers use access modifiers. We found out that libraries define more public classes than applications. Also, we observed that applications and libraries/frameworks have on average more than 25% of their classes over-exposed which could be detrimental for system modularity. Moreover, we found that around 90% of these classes are over-exposed since their first implementation. Also, we found that most over-exposed classes defined in libraries are not used by third-party applications. Moreover, we observed that while over-exposed classes are a latent risk they do not always negatively impact on the software health.

Additionally, in order to assist the detection of over-exposed classes, we have developed an Eclipse plugin that automatically identifies over-exposed classes and makes suggestions to the developer to remove unnecessary class exposure.

The article is structured as follows. Section 2 provides background information about class accessibility in Java. Section 3 analyzes and compares the accessibility modifiers of classes in plain applications and libraries/frameworks. Section 4 defines the notion of over-exposed class. Section 5 compares the number of over-exposed classes in plain applications and libraries/frameworks. Section 6 analyzes if over-exposed classes address future client requirements in libraries. Section 7 speculates on the origin of over-exposed classes and discusses their evolution. Section 8 presents our plugin for dealing with class over-exposure. Section 9 presents the threats of validity of our study. Section 10 discusses related work. Section 11 gives the conclusions and outlines future lines of work.

2. Background

This section outlines the terminology used in the article (Section 2.1) and briefly describes the Java class modifiers (Section 2.2). People familiar with Java class modifiers might safely skip Section 2.2.

2.1. Terminology related to class accessibility

We adopt the following terminology in order to ease the reading of the article. We refer to a non-nested class as a *class* or a *plain class*. We make no difference between a generic and a class since genericity is orthogonal to the concepts we are dealing with in this article. We refer to a *nested class* as a class that is syntactically embedded into a parent (static or non-static) class. A nested non-static class is often referred to as an inner class. We refer to the *encapsulating class* of an inner

class *C* as the class that contains *C*. For example, in the code `public class C1 { private class C {}}`, the public class `C1` is encapsulating the private inner class `C`.

We refer to an *accessibility modifier* as the syntactical keyword that characterizes the accessibility of a class. Four different accessibility modifiers are offered by Java (`public`, `protected`, *default*, and `private`). For example, `public` is the accessibility modifier for `A` in the code `public class A {}`. The absence of keyword represents the *default* accessibility.

We refer to *application (or plain application)* as a bundle of classes distributed to end-users. An application is not meant to be extended by third party. We refer to *library* as a bundle of classes designed to be used or extended by applications. We make no distinction between library and framework, except when relevant.

2.2. Java class accessibility

Class: A Java class can be either public or private in the package that defines it. Java provides two accessibility modifiers that can be used with a plain class:

- *Public* – A class definition preceded by the `public` modifier makes the class accessible to all other packages and classes in the system.
- *Default* – A class definition that is not preceded by an explicit modifier has the default accessibility, making the class private to the package. For example, let us consider the class definition `package p; class A {}`. Class `A` is accessible only in package `p`, and not accessible from other packages. All other classes contained in package `p`, which may be contained in the same compilation unit than `A` or not, may reference `A`. Note that `A` is not accessible to sub-packages of `p`.

Nested class: Nested classes are divided into two categories: static and non-static.[1] The terminology of Java refers as *static nested classes* to the nested classes declared static. Instead, non-static nested classes are called *inner classes*. The main difference between them is that instances of inner classes are nested inside an instance of the outer class, whose members may be accessed either directly or via the "outer this" language construction. This is not the case for static nested classes.

Java provides four accessibility modifiers to regulate the access of nested classes:

- `public`: the nested class is accessible to any class in any package (as long as the enclosing class accessibility is also public).
- `default` (no explicit modifier): the nested class is accessible only within the package of its enclosing class.
- `protected`: the nested class is accessible by classes defined in the same package and by classes that extend the enclosing class.
- `private`: the nested class is accessible by the enclosing class and other nested classes in the same enclosing class.

Additionally, Java allows the definition of classes without a name called *anonymous* classes that can be declared and instantiated at the same time. An anonymous class does not have an accessibility since it cannot be accessed. We therefore exclude anonymous classes from our analysis. Similarly, Java allows the definition of *local* classes that are nested classes contained by a block that are not a member of any class. Since local classes cannot declare an access modifier, we exclude them from our analysis.

Interfaces: An interface in Java is either public or private to its package. To keep our article focused, we exclude interfaces from our analysis.

3. Analysis of Java applications

This section assesses the distribution of class access modifiers across a collection of 30 Java applications. First, we present our benchmark (Section 3.1), and then we evaluate how access modifiers are used in practice (Section 3.2).

3.1. Benchmark

Benchmark description: We selected 30 open-source Java systems and measured the use of access modifiers for classes. These systems and the results of the metrics relevant to our study are listed in the appendix. Some of these systems were chosen because of their popularity among the community of developers. In addition, these are the same systems that were subject of our previous research study of methods accessibility [4]. The appendix lists all the software versions to let others easily reproduce our study. Additionally, the datasets[2] and the processing code[3] used to conduct the experiments are available for download. A total of 15 of these systems are libraries or frameworks while the remaining 15 are plain

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
[2] http://bit.ly/mseFiles
[3] http://ss3.gemstone.com/ss/SPIRIT.html (package Spirit-ExhibitionismTests)

Table 1
Classes analyzed.

Kind of applications	#Plain	#Static nested	#Non-static nested	#Total
Libraries/frameworks	13,536 (81.97%)	2173 (13.16%)	804 (4.87%)	16,513
Plain applications	10,550 (77.19%)	1435 (10.5%)	1682 (12.31%)	13,667
Total	24,086 (79.81%)	3608 (11.95%)	2486 (8.24%)	30,180

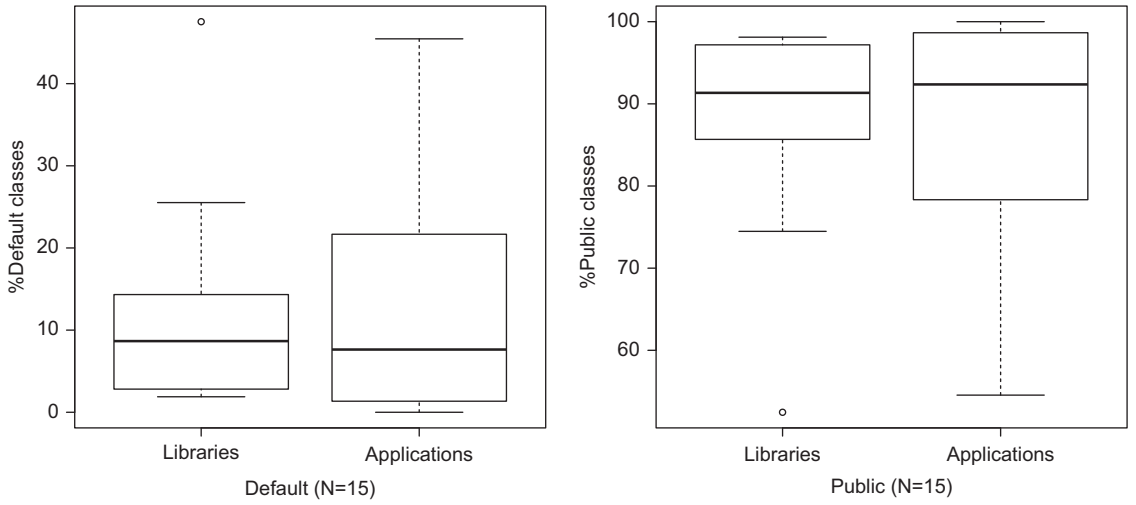

Fig. 1. Distribution of plain classes accessibility.

applications. We distinguish libraries and frameworks from plain application because these two kinds of systems have different goals. The libraries and frameworks[4] are meant to be extended, used, and/or instantiated by an application. On the contrary, the primary goal of a plain application is not to be extended or used by a third application. As we will verify subsequently, these differences lead to a different usage of the access modifiers.

Analyzing the benchmark: We used the Moose[5] software analysis platform to carry out our analysis. The Java source code of the 30 systems was obtained from public online archives. All non-Java files were excluded from our analysis. Then, using Moose, we statically analyzed the source code of the systems. In the cases of systems with unit tests, we included the tests in our analysis. Specifically, 8 applications and 11 libraries/frameworks contain unit tests. All our measurements are reported with a precision of 0.01%, meaning that we rounded up the values to the second decimal place.

3.2. Accessibility distribution

Our benchmark totals 30,180 classes and nested classes (Table 1). Non-nested classes defined in libraries represent 81.97% of the classes while this value is 77.19% in plain applications. Nested classes represent the 18.03% (73% of them are static) and 22.81% (46% of them are static) for libraries and applications respectively.

Given the different goals of libraries and applications (see Section 3.1), and based on RQ1, we hypothesize that applications declare less public classes than libraries. We analyze the distribution of the different accessibility modifiers for the classes of the libraries and applications. We distinguish the accessibility for plain classes and nested classes. The results for each application are listed in Tables A.8 and A.11.

Fig. 1 plots the distribution of plain classes with a given accessibility regarding the total number of plain classes for the 15 applications and 15 libraries ($N=15$). Both libraries and applications define more public classes than default ones. On average, in libraries 11.78% of classes are default and 88.22% are public. In applications, 13.65% of classes are default and 86.35% are public. In order to answer RQ1, we stated the following null hypothesis:

- $H1_0$: there is no difference between plain applications and libraries on how frequently they define plain public classes.

First, we tested the data for normality using the Shapiro–Wilks test. Since we found that the data is normally distributed ($p-value = 0.5279$), we used the t-test to check if there is a significant difference between the distribution of public classes

[4] For the sake of simplicity, the words “library” and “framework” are used indistinctly in the rest of the article.
[5] http://www.moosetechnology.org/

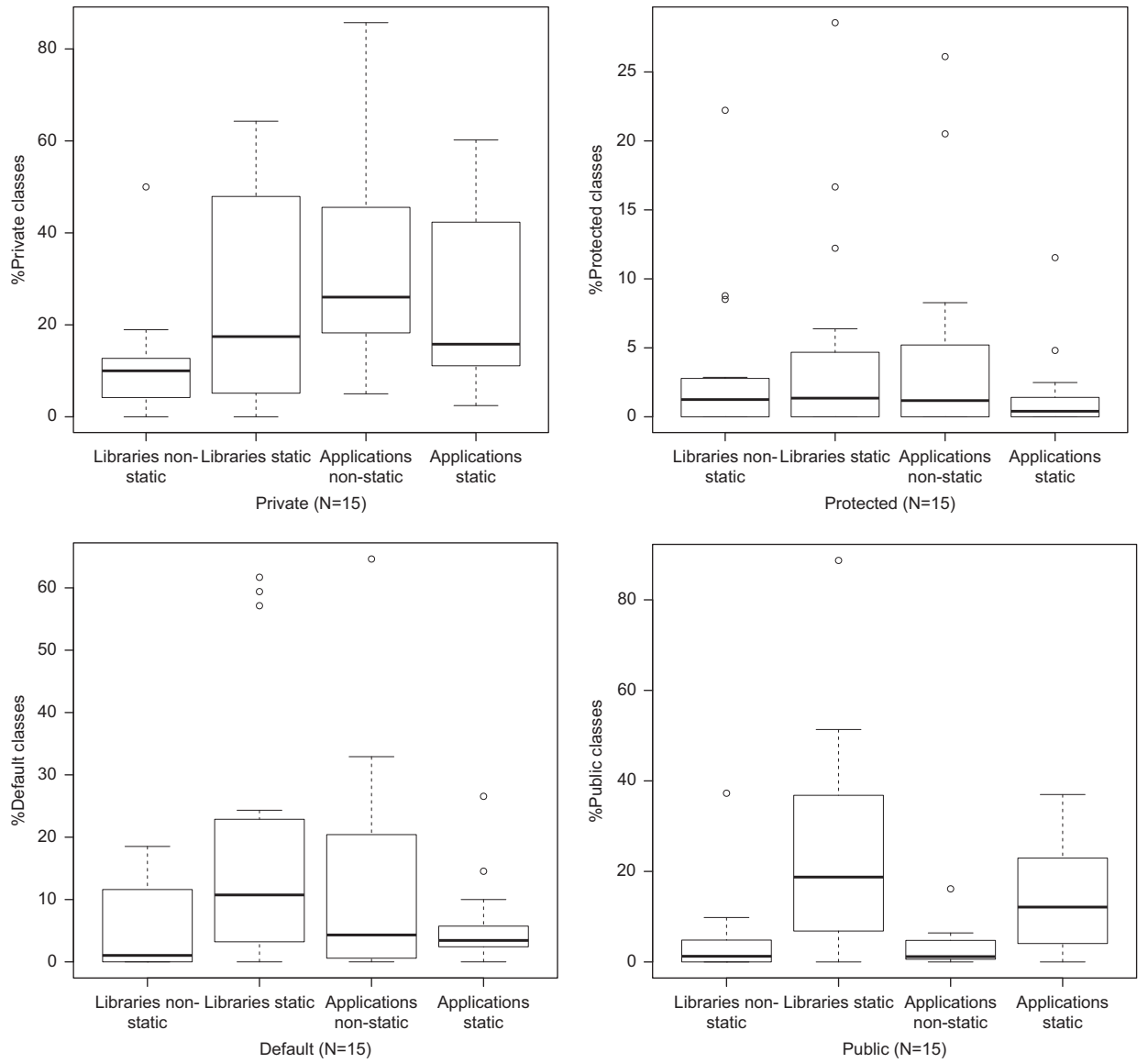

Fig. 2. Distribution of nested classes accessibility.

defined in libraries and plain applications. After testing we rejected $H1_0$ with $\alpha = 0.05$ and $p{-}value = 0.01031$ indicating that the two distributions are statistically different. We then concluded that *libraries have on average significantly more public classes than plain applications*. This confirms our intuition that libraries should expose more classes than applications since they are designed to be extended by third parties.

Regarding the accessibility defined for nested classes, we distinguish static from non-static nested classes. Fig. 2 plots the ratio of nested classes with a given accessibility regarding the total number of nested classes.

Unlike the case of non-nested classes, few non-static nested classes are public on average. The `public` access modifier is the least used in non-static nested classes in both, libraries (4.8%) and applications (3.26%). These results would indicate that developers frequently create inner classes manually as internal data structures that are intended to be only used by the parent class. These values are higher for static nested classes: 25.83% and 14.17% respectively. We define two null-hypotheses to verify the statistical significance in the data we gathered:

- $H2_0$: there is no difference between plain applications and libraries on how frequently they define non-static nested public classes.
- $H3_0$: there is no difference between plain applications and libraries on how frequently they define static nested public classes.

We found that in both cases the data deviate from normality using Shapiro–Wilks test (p-values: 1.817^{-7} and 2.201^{-9}). For this reason, we use the Mann–Whitney-test to check a statistical difference. This test shows that $H2_0$ and $H3_0$ cannot be rejected using a two-tailed test with a probability of error (or significance level) $\alpha = 0.05$. This conclusion is based on the ranks values calculated by the test: $T_1 = 238$, $U_1 = 107$, $U'_1 = 118$, $T_2 = 252.5$, $U_2 = 92.5$, $U'_2 = 132.5$ [6]. This means that there is no statistical significant difference on how libraries and applications define non-static and static nested public classes.

Regarding the classes defined as default, we found different values for libraries and applications. 5.29% of the nested non-static classes defined in libraries are default. This value is higher in applications: 12.62%. When it comes to nested static classes, they represent the 18.92% in libraries and the 5.79% in applications.

We observed an interesting fact here: the protected accessibility is rarely used in nested classes despite being heavily used in methods [4]. Non-static classes represent on average 3.44% of the total number of nested classes in libraries and 4.71% in applications. These values for static classes are 4.87% and 1.61% respectively.

The private accessibility modifier is the most used one with nested classes. Libraries define on average 11.49% of non-static nested classes and 23.37% of static ones. Plain applications report 33.02% on average of non-static nested classes and 24.83% of static ones.

In summary, to answer RQ1, libraries define more public plain classes than applications. However, libraries and applications define similar percentages of public nested classes.

4. Class over-exposure

Each class of a Java system is defined with an access modifier. Class accessibilities define the interface of packages. We have discovered that it frequently happens that a class is over-exposed by having an accessibility greater than the strictly necessary. We identify this situation as a code anomaly [7] that should be considered by the developer. For instance, let us consider the situation shown in Fig. 3 from one of our systems. `HomeFrameController` is a public class that is referenced *solely* in the package in which it is defined. Such a class could therefore be default (i.e. private to its package) instead of being public. We qualify this class as *over-exposed* since its accessibility is too broad against its usage. By reference we mean explicit declarations of a class (e.g. declaring a variable of a given class inside a method) but also implicit references to a class (e.g., method calls involving classes). For example, let us consider a method `foo()` defined in class `A`, and a method `bar()` defined in class `B` that returns an instance of A. Given a method in a class `C` with the invocation `new B().bar().foo()`, we say that class `C` references classes `B` and `A`. Note that class `B` is explicitly referenced while class `A` is implicitly referenced.

Certainly, there are cases where a class is voluntarily over-exposed by developers. For example, the developer might want to prepare the system for a likely future evolution, to write a unit test or to enable third-party applications to use the over-exposed class. We carefully consider this situation in Section 6. In this section, we list the criteria and code patterns for marking a class as being over-exposed, according to the class type. Remember that these criteria are based on information provided by a static analysis of the code.

Public plain class: A public plain class *C* is over-exposed if *C* is only referenced by classes defined in the same package. The adequate accessibility is private, using the default accessibility modifier.

Public inner-class: A public inner-class *C* is over-exposed if:

- *C* is only referenced by the enclosing class and/or other inner classes defined in the same enclosing class. The adequate accessibility is private.
- *C* is only referenced by classes defined in the same package and classes that extend the enclosing class (and at least one is defined in other package). The adequate accessibility is protected.
- *C* is only referenced by classes defined in the same package. The adequate accessibility is default.

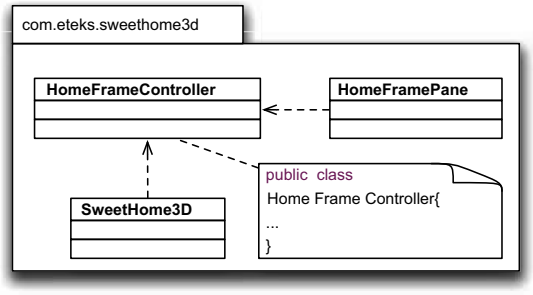

Fig. 3. Example of over-exposure in class HomeFrameController.

Default-accessible nested class: A nested class *C* is over-exposed if *C* is only referenced by the enclosing class and/or sibling nested classes (i.e. another nested class in the same enclosing class). The adequate accessibility is private.

Protected nested class: A protected nested class *C* is overexposed if:

- *C* is only referenced by the enclosing class and/or other sibling nested classes. The adequate accessibility is private.
- *C* is only referenced by classes in the same package. The adequate accessibility is default.

Note that Java only permits a class to be protected if it is nested. Non-nested classes cannot be protected.

Private class: A private class (i.e. default accessibility if non-nested and `private` accessibility if nested class) cannot be over-exposed since being private to its package is the most restrictive accessibility.

5. Measuring class over-exposure

This section gives our measurements of over-exposure in our benchmark.

Analyzable classes: Not all classes can be considered as over-exposed. In particular, there are two situations in which the over-exposure of classes cannot be determined:

1. *Classes whose accessibility cannot be changed*: Anonymous classes do not have an explicit accessibility modifier, thus, its accessibility cannot be changed. Also, non-nested default classes and nested private classes are in this situation since their accessibility cannot be reduced.
2. *Unreferenced classes*: Classes that are not directly referenced by any class (class declaration, field declaration, parameters, etc.) in the application cannot be determined. This means that we cannot tell whether the class is over-exposed. In this case, the class could be dead code, used via reflection or solely used by third-party applications. We observed that 30.68% (5066) of classes in libraries and 17.63% (2409) of classes in applications are not directly referenced by any class.

Libraries: Table 2 shows the results of over-exposure in libraries. The table only shows the results for classes whose accessibility could be over-exposed (i.e. non-nested default and nested private classes are not shown). While libraries defined a total of 16,513 classes only 14,790 of them could be over-exposed. From these 14,790 classes, 66.24% of them are analyzable. The *Right accessibility* column indicates the number of classes that have an adequate accessibility (i.e., the accessibility that is strictly necessary). On the contrary, the *Over-exposed* column indicates the number of classes that fit into one of the patterns described in Section 4 and whose accessibility can be restricted.

From 9797 analyzable classes, 4433 have an adequate accessibility meaning that 5364 classes are over-exposed. In total, 32.48% (5364 of 16,513) of all the classes defined in libraries are over-exposed (36.27% of the classes that could be over-exposed). However, it is important to remark that these values should be carefully viewed because the percentage of over-exposed classes could be over-estimated. For instance, some public classes that are currently invoked only from inside their packages could have been designed to be invoked by third-party applications.

For the total number of classes, the range of over-exposed classes defined by libraries goes from 8.23% to 53.58%, with an average of 28.5%, and a standard deviation of 11.81%. After carefully analyzing the source code of the library with the lowest percentage of over-exposure, JFreeChart (8.23%), we found that it has the highest percentage of non-nested public classes 93.25% (see Table A1). We interpret here that the low number of over-exposed classes of JFreeChart is not an indicator that developers restricted the access modifiers on purpose but rather that a high number of public classes are called outside their packages.

Interestingly, we found that 65.4% ($806+388+207=1{,}401$ of 2142) of the nested classes that could have an accessibility broader than necessary are over-exposed (47.06% of the total number of nested classes in libraries). Moreover, 73.89% (3963 of 5364) of the over-exposed classes are plain public classes. This is shown in Table 3 that details the changes that should be made to restrict the accessibilities of the classes to their minimum necessary. Regarding nested classes, most of the accessibility of them should be changed to private ($10.38\% + 3.11\% + 7.23\% = 20.72\%$).

Table 2
Over-expose results for libraries/frameworks.

Classes	Defined	Analyzable	Right accessibility	Over-exposed
Plain public	12,648	8247 (65.2%)	4284 (33.87%)	3963 (31.33%)
Non-nested default	888	825	825	0
Nested public	1440	918 (63.75%)	112 (7.78%)	806 (55.97%)
Nested default	492	424 (86.18%)	36 (7.32%)	388 (78.86%)
Nested protected	210	208 (99.05%)	1 (0.48%)	207 (98.57%)
Total	14,790	9797 (66.24%)	4433 (29.97%)	5364 (36.27%)

Table 3
Suggested accessibilities for libraries.

Current accessibility	Suggested accessibility	# of classes over-exposed	% of classes over-exposed
Plain public	Plain default	3963	73.89
Nested public	Nested default	223	4.16
Nested public	Nested protected	26	0.48
Nested public	Nested private	557	10.38
Nested protected	Nested default	40	0.75
Nested protected	Nested private	167	3.11
Nested default	Nested private	388	7.23
Total		5364	100

Table 4
Over-exposure results for applications.

Classes	Defined	Analyzable	Right accessibility	Over-exposed
Plain public	9262	7288 (78.69%)	4683 (50.56%)	2605 (28.13%)
Non-nested default	888	825	825	0
Nested public	491	386 (78.62%)	134 (27.29%)	252 (51.32%)
Nested default	657	586 (89.19%)	87 (13.24%)	499 (75.95%)
Nested protected	283	277 (97.88%)	8 (2.83%)	269 (95.05%)
Total	10,693	8537 (79.84%)	4912 (45.94%)	3625 (33.9%)

While libraries and frameworks are implemented to be extended or used by other applications, they have some differences. For example, frameworks have a low number of classes with extension points when compared to the number of classes of libraries that are meant to be used directly by clients. For this reason, we checked for any statistical difference in the over-exposed values of libraries and frameworks. Our collection of applications only contains 4 frameworks, namely: Hibernate, JHotDraw, JUnit, and Struts. The remaining 11 systems are libraries. After checking the normality of the data, using the Shapiro–Wilks test ($p-value = 0.9746$), we used the t-test to check for any significant difference. After testing, we could not reject the hypothesis that libraries and frameworks define different percentages of over-exposed classes ($\alpha = 0.05$ and $p-value = 0.6038$). Thus, we concluded that libraries and frameworks have similar percentages of over-exposed classes.

Applications: Table 4 shows the results of over-exposure for plain applications. The analyzed applications define a total of 13,667 classes from which 10,693 of them that could be over-exposed. From these 10,693 classes, the 79.84% of them are analyzable.

We found that 26.52% (3625 of 13,667) of all the classes present in applications are over-exposed (33.9% of the classes that could be over-exposed). The percentage of over-exposed classes found in applications goes from 7.82% to 37.16% with an average of 25.44% and a standard deviation of 9.73%. In this case, Logisim and Portecle are the applications with the lowest percentages of over-exposed classes (7.82% and 8.16%). After manually reviewing their source code, we found evidences that developers carefully restricted access modifiers. Both applications present the highest percentages of non-nested default classes (24.54% and 35.68%).

Similarly to the case of libraries, 71.85% (2605 of 3625) of the over-exposed classes are plain public classes whose accessibility should be changed to default (Table 5). Regarding the nested classes, 71.28% ($252+499+269=1020$ of 1431) of the classes that could have an accessibility broader than necessary are over-exposed (32.72% of the total number of nested classes defined in applications). Moreover, as in the case of libraries, most over-exposed nested classes accessibility may be changed to private (23.65% of the total number of over-exposed classes).

Relationship with over-exposed methods: This article is about class over-exposure. In a previous work [4] we studied over-exposure for methods. Contrasting these two levels of granularity (class and method) is relevant to understand the cause of over-exposure. Specifically, we wanted to investigate whether an over-exposed class has more over-exposed methods than classes with adequate accessibility.

We computed the number of over-exposed methods for each over-exposed and non-nested class. For example, a class with 9 non-overexposed methods and 1 over-exposed method has a ratio of 10% of method over-exposure. We computed this metric for each library and plain application. We compare the result with that of non-over-exposed classes. We found that, on average, over-exposed classes of libraries have 11.38% of over-exposed methods (standard dev. of 3.5%). Non-over-exposed classes have a ratio of 6.65% over-exposed methods (standard dev. 2.62%). Moreover, we consistently found that in each library over-exposed classes have around twice the percentage of method over-exposure than non-over-exposed classes. For plain applications, over-exposed classes have on average 11.11% of over-exposed methods (with a standard deviation of 4.67%), while non-over-exposed classes have 5.99% of over-exposed methods (standard dev. of 3.54%). Similarly to the case of libraries, we consistently found that in each application over-exposed classes define twice the number of over-exposed methods than non-over-exposed classes.

Table 5
Suggested accessibilities for applications.

Current accessibility	Suggested accessibility	# of classes over-exposed	% of classes over-exposed
Plain public	Plain default	2605	71.85
Nested public	Nested default	97	2.68
Nested public	Nested protected	9	0.25
Nested public	Nested private	146	4.03
Nested protected	Nested default	57	1.57
Nested protected	Nested private	212	5.85
Nested default	Nested private	499	13.77
Total		3625	100

The ratios were similar when we analyzed the classes having at least one over-exposed method. In the case of libraries, 50.71% of the over-exposed classes have at least one over-exposed method (with a standard deviation of 11.67%). However, only 24.64% of the non-over-exposed classes have at least one over-exposed method (with a standard deviation of 9.93%). We obtained similar results for applications. On average, 46.96% of over-exposed classes have at least one over-exposed method (with a standard deviation of 11.67%). This value is lower for non-over-exposed classes: 22.68% (with a standard deviation of 8.34%).

While the percentage of method over-exposure in classes is not really high, over-exposed classes have on average twice as much over-exposed methods than non-over-exposed classes. We hypothesize that this situation happens because a percentage of over-exposed classes are prepared to be used by other classes.

Comparison: Overall, we found that around 35% of the analyzable classes in libraries and applications are over-exposed. For this reason, *we answer RQ2 as follows: there is a significant portion of classes over-exposed. Moreover, in both cases more than 70% of the over-exposed classes found are due to non-nested classes defined as public that should be default.* As we mentioned before, we think that some over-exposed classes in libraries were left on purpose to be extended or used by other applications. For this reason, we hypothesized that libraries should have more over-exposed classes than plain applications. A statistical test is necessary to support our claims. First, we tested the data for normality using the Shapiro–Wilks test. We obtained $p-value = 0.3479 > 0.05$ indicating the data is normal. For this reason, we can use Student's *t*-test to check for any statistical difference of over-exposed classes in libraries and plain applications. We stated our null hypothesis as follows:

- $H4_0$: libraries have the same proportion of over-exposed classes as applications have.

After testing we can reject $H4_0$ with $\alpha = 0.05$ and $p-value = 7.225^{-14}$ indicating that there is enough statistical evidence to suggest that libraries define more over-exposed classes than applications.

A special case that should be mentioned here is the existence of internal implementation packages. Specifically, Eclipse has adopted an “internal” naming convention[6] in which public classes hosted by a package having “internal” in its name must not be accessed by clients. That is, classes in this kind of packages should not be considered as over-exposed. We checked the analyzed systems and found that only 5 of them have packages that follow this convention, namely: Hibernate, JUnit, Maven, Argo, and FindBugs. However, this convention only appeared in 3.3%, 0.85%, 7.91%, 0.8%, and 0% of the total number of over-exposed classes found in each system. For this reason, we argue that this situation is not really important for our analysis.

Incidence of JUnit test on accessibility: We analyzed the incidence of JUnit test on the accessibility of plain classes as it may introduce a bias. Since classes accessed by tests must be public, this could lead to classes with a broader accessibility than the one originally intended by the developer. However, classes only called outside their packages by tests are not over-exposed (their accessibility must be public because they are actually called by a class outside their package). Since we include tests classes in our analysis, we are not considering these classes as over-exposed.

As it was mentioned before, 8 applications and 11 libraries/frameworks of our study contain unit tests. After analyzing these systems, we found that on average the percentage of plain public classes in libraries that are only referenced outside their packages by tests is 6.28% (with a standard deviation of 6.15%). The same value was computed for applications and turned out to be smaller: 2.61% (with a standard deviation of 2.28%). As it was expected, the percentage is higher in libraries, since libraries define more classes that are only going to be externally referenced by third-party applications (or, in this case, by tests). Taking into account these results, we can conclude that the incidence of JUnit test on accessibility is small. The detailed results for each system are listed in Table A10 and A13 in the Appendix.

[6] https://wiki.eclipse.org/Naming_Conventions

Table 6
Analysis of library usage (by external applications).

Library	#Applications analyzed	#Public classes in library	#Over-exposed public classes in library	#Over-exposed public classes called externally by applications	#Over-exposed public classes taking into account applications
Commons-Compress	7	142	39	8	31
Javassist	7	239	84	15	69
Jericho-HTML	4	65	12	1	11
JFreechart	5	891	41	6	35

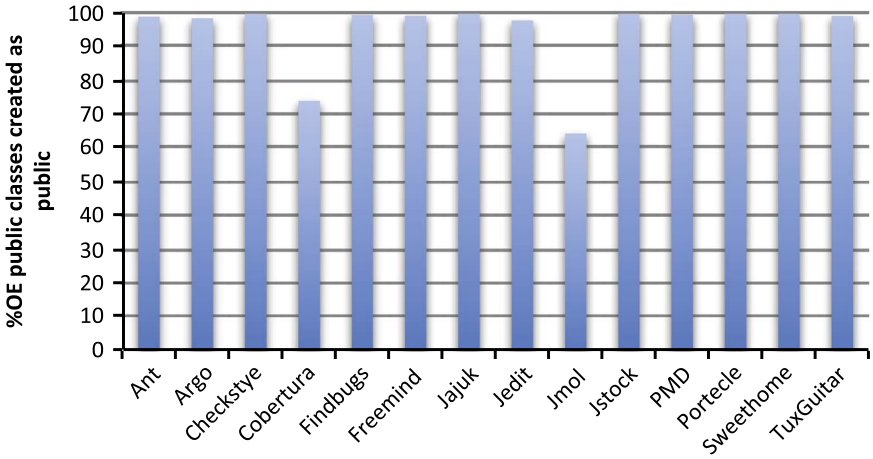

Fig. 4. Analysis of the creation of over-exposed public classes.

6. Analyzing the use of libraries

Since the purpose of a library is to be used by others, it is expected that classes belonging to a library will have an accessibility broader than necessary. In this case, reducing the accessibility of classes could inhibit external classes from referencing the class. Thus, it is relevant to measure the number of over-exposed classes when the libraries are used by external applications. In this way, we expect to answer RQ3 by analyzing whether over-exposed classes address future client requirements.

We used the following methodology: (i) we identified a number of libraries being relevant to be part of this experiment; (ii) for each library, we selected a number of applications that use the library; (iii) we measured the number of over-exposed classes; (iv) we deduced whether over-exposed classes defined in libraries actually met the need of applications using the libraries.

Among the 15 libraries that are part of our benchmark, we picked up 4 that had a reasonable amount of client applications. These libraries are Commons-Compress, Javassist, Jericho-HTML, and JFreechart. For each of these libraries, we have between 4 and 7 client applications. These applications are part of the Maven repository.[7] Note that Maven provides information about applications that use the same version of the library. This information facilitated our identification of external applications.

Table 6 measures the number of over-exposed classes when considering client applications. For each library, we check if one or more of their over-exposed public classes found in Section 5 are referenced in the external applications that use them. Having a class identified as over-exposed in the library that is referenced by an external application means that the class is not over-exposed in the presence of such an application.

We analyzed 7 external applications that use Commons-Compress. We found that 27.46% (=39) of the public classes defined by Commons-Compress were over-exposed (when no external applications are considered). However, after analyzing the external classes, we found that 8 classes identified as over-exposed are referenced externally. In this way, there are still 31 classes over-exposed. That is, most over-exposed classes (79.49%) seem not to address client requirements. We observed that this percentage is similar for the remaining libraries: 82.14%, 91.67%, and 85.37% for Javassist, Jericho-HTML and JFreechart, respectively. That is, for this sample of applications, most over-exposed classes remain over-exposed even in the presence of external client applications.

[7] http://mvnrepository.com/

By analyzing these results we can answer RQ3 as follows: Only a small portion of over-exposed classes identified in libraries address client application usage. However, in order to claim that the values of over-exposure for public classes defined in libraries are valid, the analysis needs to be extended to a larger number of client applications. This study will be carried out in future work.

7. Creation and evolution of over-exposed classes

What make developers create so many over-exposed classes? We investigate this non-trivial question by monitoring over-exposed classes over multiple revisions of software systems. We restricted our analysis to (i) plain applications and (ii) public classes since public classes concentrate most of the over-exposure. Only one version is available for the Logisim application, we therefore exclude this application and report the evolution of the 14 remaining applications. We analyzed a set of versions of each application since the earliest version.

7.1. Creation of over-exposed classes

Specifically, we analyzed if the plain public classes being over-exposed in the last version of each application were created with the public accessibility modifier. In order to do this, we looked at the version history to find the version in which an over-exposed class appeared for the first time. Fig. 4 reports the ratio of classes over-exposed in each application that were created with public accessibility. A total of 13 applications, from the 14 analyzed, have more than 90% of their over-exposed non-nested classes created as public. Only two applications (Cobertura and Jmol) showed lower values (73% and 64%). After analyzing these 2 applications, we found that they have a larger number of non-nested default classes than the other applications at early versions, which could be the cause of these outliers. From a total of 2113 over-exposed classes analyzed, 2043 were created as public and were over-exposed. *We answer to RQ4 (Are classes over-exposed since their first implementation, or do they become so over time?) affirmatively because 96.69% of the over-exposed public classes have an access modifier broader than necessary since their creation*. This is an interesting fact since many IDEs, such as Eclipse or NetBeans, use *public* as the accessibility by *default* to create classes. While analyzing how the IDEs influence the creation of over-exposed classes could be interesting to understand the phenomenon, it is out of the scope of this article and it will be analyzed in future work.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
a: ant																		X	
b: check			X																
c: coveragedata		X			X								X	X	X		X	X	
d: coveragedata.countermaps			X																
e: instrument			X																
f: javancss							X	X	X	X	X			X					
g: javancss.ccl						X													
h: javancss.parser						X													
i: javancss.parser.debug						X													
j: javancss.parser.java15						X													
k: javancss.parser.java15.debug						X													
l: javancss.test																			
m: merge			X																
n: reporting			X			X									X		X	X	
o: reporting.hlml			X											X				X	
p: reporting.html.files																			
q: reporting.xml			X											X				X	
r: util	X													X	X		X		
s: webapp																			

Density: 11%;
Propagation cost: 0.22;
Cyclicality: 0.53;
Stability: 76%

Fig. 5. Design structure matrix of Cobertura (e.g. package *coveragedata* depends on package *util*).

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
a: ant		O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	X	O
b: check	O		X	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
c: coveragedata	O	X		O	X	O	O	O	O	O	O	O	X	X	X	O	X	X	O
d: coveragedata.countermaps			X																
e: instrument	O	O	X	O		O	O	O	O	O	O	O	O	O	O	O	O	O	O
f: javancss							X	X	X	X	X			X					
g: javancss.ccl						X													
h: javancss.parser	O	O	O	O	O	X	O		O	O	O	O	O	O	O	O	O	O	O
i: javancss.parser.debug	O	O	O	O	O	X	O	O		O	O	O	O	O	O	O	O	O	O
j: javancss.parser.java15	O	O	O	O	O	X	O	O	O		O	O	O	O	O	O	O	O	O
k: javancss.parser.java15.debug	O	O	O	O	O	X	O	O	O	O		O	O	O	O	O	O	O	O
l: javancss.test																			
m: merge	O	O	X	O	O	O	O	O	O	O	O	O		O	O	O	O	O	O
n: reporting	O	O	X	O	O	X	O	O	O	O	O	O	O		X	O	X	X	O
o: reporting.hlml	O	O	X	O	O	O	O	O	O	O	O	O	O	X		O	O	X	O
p: reporting.html.files																			
q: reporting.xml			X											X				X	
r: util	X													X	X		X		
s: webapp																			

Density: 58%;
Propagation cost: 0.89;
Cyclicality: 0.89;
Stability: 15%

Fig. 6. A worst-case design structure matrix of Cobertura when considering over-exposed classes ('X' means existing dependencies and 'O' unnecessary class exposure).

7.2. Speculative analysis of evolution of over-exposed classes

Another interesting question is to understand how over-exposed classes will affect the system in the future. With this goal in mind, we performed a speculative analysis of dependencies for Cobertura. In many Java systems, it is reasonable to assume that packages reflect architectural components, and thus, we can model the (static) system architecture in terms of these components and their dependencies. In particular, Fig. 5 shows Cobertura as a DSM (Design Structure Matrix) [8] in which the rows/columns represent Java packages (i.e., components) of the application (columns names are not shown for simplicity, they are the same than the rows). Given a cell for components A (row) and B (column), an 'X' in the cell means that there is at least one access from A to B being realized via some class. For example, package *coveragedata* depends on (i.e., makes use of classes of) 7 other packages, including check, instrument, and merge. The components shaded in gray are those in which the over-exposed classes are located.

The resulting matrix[8] may be used to assess component modularity, by measuring properties such as propagation cost, density, or number of cycles, or stability, among others [9]. These metrics were computed with the help of the Lattix DSM[9] tool. *Propagation cost* measures the extent to which a change in one element impacts other elements; and it is a function of the degree of coupling among elements (computation details can be found in [10]). Propagation cost is an important indicator of a system overall architectural complexity. This numerical indicator ranges from 0.0 to 1.0. A value close to 0 reflects a good modularization and low propagation of changes. Conversely, a value close to 1 means that a change in a component can propagate to almost all components. Cyclicality is a measure of how many components are involved in (at least) a cycle, since cycles are often correlated with bugs and maintenance problems. The closer cyclicality is to 1.0, the more cycles the system has. Stability, in turn, assesses how much of the system is affected when a change is made. A system with low stability (closer to 0.0) is generally fragile even to small changes. The density of the DSM is the ratio of the number of dependencies in the matrix and the number of all possible dependencies. The values of these metrics for Cobertura based on the (actual) DSM are given in Fig. 5.

Having an over-exposed (public, non-nested) class C defined in a package means that, in principle, all other packages in the system might access class C. For instance, a developer might have inadvertently created C with a public access modifier, because she used an IDE with `public` as the default option. Anyway, from the perspective of modularity, class C can produce a rippling effect in the system that enables extra dependencies among components. This effect is likely to increase coupling and cycles in the system [11,12]. Many of these unwanted dependencies might be also not prescribed by the original architecture design. Fig. 6 shows a worst-

[8] The matrix can include additional dependency information (e.g., variable access, method calls, etc.), which is not considered in our analysis.
[9] http://www.lattix.com

Table 7
Analysis of evolution of over-exposed classes.

Application	#Over-exposed classes in v1	%Over-exposed classes in v1 that remain over-exposed in v2	%Over-exposed classes in v1 fixed in v2 by calls	%Over-exposed classes in v1 fixed in v2 by changing the accessibility	%Over-exposed classes in v1 deleted in v2
Ant	367	83.11	5.18	0	11.71
Argo	593	37.94	18.72	6.58	36.76
Checkstyle	133	81.95	0	0	18.05
Cobertura	24	75	0	0	25
Findbugs	336	92.86	1.19	0.3	5.65
Freemind	45	71.11	15.56	0	13.33
Jajuk	186	100	0	0	0
JEdit	343	93	1.46	0.88	4.66
JMol	142	13.38	1.41	2.82	82.39
JStock	88	55.68	17.05	0	27.27
PMD	158	96.84	2.53	0	0.63
Portecle	9	88.89	11.11	0	0
SweetHome3D	64	93.75	3.125	3.125	0
TuxGuitar	25	4	0	0	96
Average		69.57	5.55	1.05	23.83
Standard deviation		31.08	6.95	1.91	30.26

case DSM for such dependencies, in which the over-exposed classes of Cobertura end up accessed by all possible components. Under this scenario, the density of dependencies in the DSM goes from 11% to 58%, which represents a significant coupling increase with negative effects on system modularity. The over-exposed classes precipitate an increase to 0.89 in the propagation cost (the initial propagation cost was 0.22), and also lead to a higher cyclicality (the initial value of 0.53 becomes 0.89). The design stability consequently goes down from 76% to 15%, as an indicator of the modularity degradation. Note that the rippling effect (marked by cells with 'O' in the figure) depends on the distribution of the over-exposed classes across components, as well as on the initial interactions (coupling) among the components. Certainly, the average case of degradation might be not as severe as in Fig. 6 (or even some new dependencies might be justified by architectural rules). For example, in a realistic scenario, the amount of dependencies caused by over-exposed classes ('O') can be lower than in Fig. 6. Thus, the global propagation cost of Cobertura should fluctuate between 0.22 and 0.89 in an average case. The DSM of Fig. A1 (Appendix) shows an average case of added package dependencies computed using link prediction techniques from social network analysis.[10] Note that the metrics for propagation cost, stability and cyclicality have moderate values, but higher than those of the initial DSM for Cobertura. Nonetheless, we believe this example is representative of the risk of having many over-exposed classes as they might hinder system modularity.

7.3. Analysis of evolution of over-exposed classes in history

While the previous analysis shows the latent risk of having over-exposed classes, it is necessary to analyze the evolution of these classes over time. To do so, we took the first and last versions of the same 14 applications described before. Then, we identified the over-exposed classes in the first version (v1) and analyzed if they were still over-exposed in the last version (v2). There are four possible situations, namely:

1. The class remains over-exposed. That is, there were not new references that affected the over-exposure of the class.
2. The class stopped being over-exposed due to a new reference from other class. For example, in the case of a non-nested public class, a reference from a class defined outside its package. This is the case in which an over-exposed class can degrade the modifiability of the system.
3. The class stopped being over-exposed because its accessibility was reduced to the minimum necessary.
4. The class was deleted or moved to other package (the latter case cannot be identified automatically since there may be more than one class with the same name).

Table 7 shows the results. On average only 5.55% of the over-exposed classes in v1 are not over-exposed in v2 because of new references. Moreover, on average, only 1.05% of the over-exposed classes in v1 are not longer over-exposed in v2 because its accessibility was reduced to the strictly necessary. In fact, most of the over-exposed classes in v1 remain over-exposed in v2 (69.57%) or were deleted (23.83%). In the cases of JMol and TuxGuitar, which present a high percentage of deleted classes, we manually analyzed the source code. We observed that more of 90% of the classes were not deleted in v2 but moved to a different package. After manually checking that these were the same classes that existed in v1, we found that more than 95% of them were still over-exposed.

[10] http://be.amazd.com/link-prediction

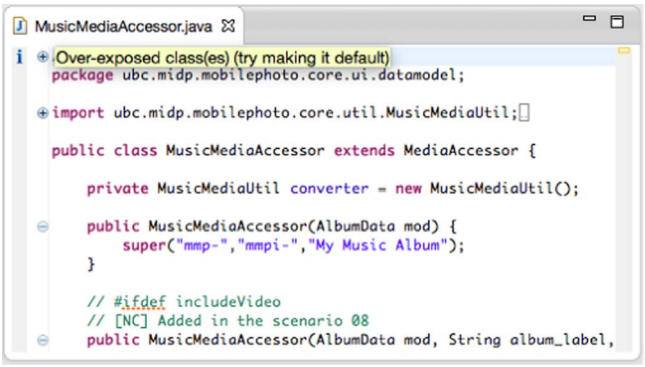

Fig. 7. Refactoring suggestion of Cover.

Also, we analyzed the evolution of over-exposed classes in 5 revisions between the first and last versions. Specifically, we looked for a pattern in which over-exposed classes in v1 stopped being over-exposed by new references (case 2), but eventually they were fixed. That is, a developer removed the reference. This situation could mean that the class was not designed to be accessed with that level of accessibility. We found this pattern in only 3 of 14 applications: Argo (19.73% of the over-exposed classes of v1), Findbugs (0.3%), and Freemind (2.22%). Then, we conclude that the pattern rarely appears. In fact, from a total of 2513 over-exposed classes found in the first versions of the 14 applications, 2161 of them remain over-exposed (without intermediate changes) until the last version or its deletion (86%). Moreover, when we analyzed the over-exposed classes of v1 that stopped being over-exposed at least in one of the versions, we found that this situation occurred in only 289 of the classes (11.5%).

Summary: The fact that over-exposed classes remain over-exposed after a number of versions suggests that over-exposed classes are not a perceived problem. While the accessibility of classes is broader than necessary, the existence of new references from classes outside the "minimal" accessibility are unusual. In this way, *we can answer RQ5 by saying that over-exposed classes do not usually negatively impact on the software health*. That is, over-exposed classes are not a major contributor to the erosion of the public API and the modularity problems associated to them. This result is interesting because it would indicate that developers are not usually aware of the implications of access modifiers of classes. However, according to classical textbook on software encapsulation [13–16], over-exposed classes are a potential problem if they are called from classes outside their intended scope. This could be specially harmful in libraries since it will introduce incompatibilities in the API [11,12].

8. Automated detection in IDE

With the goal of making developers aware of the existence of over-exposed classes we extended *Cover*,[11] an Eclipse plugin that detects over-exposed methods, to also detect over-exposed classes. The usage of this plugin would also help to cope with the degradation symptoms analyzed in Section 7 that might affect system modularity.

The plugin uses the Java Development Tools of Eclipse (JDT) to iterate over all the classes and retrieve the calling classes for each class of the application. Then, *Cover* lists the over-exposed classes (structured along packages), and for each over-exposed class, the plugin indicates the most appropriate accessibility for the current state of the application (Fig. 7). To determine whether a class is over-exposed we implemented in Cover the patterns described in Section 4.

To evaluate our plugin, we applied it to SweetHome3D,[12] one of the applications of our benchmark. We followed five steps:

1. run the application and try out the tutorial;
2. automatically find the over-exposed classes using *Cover*;
3. reduce the accessibility of each over-exposed class to the one suggested by the plugin (i.e. its strict necessary accessibility);
4. recompile SweetHome3D;
5. run all the tests and verify that they all pass;
6. run the tutorial and look for odd behavior.

We successfully conducted this experiment using SweetHome3D. After refactoring the source code, the application compiled and we did not notice any odd or unexpected behavior when the application was re-run. These steps can be easily applied to other applications as well. Our refactoring of SweetHome3D is available at https://db.tt/xr9LtZOC. The percentage of over-exposed classes detected was 16.92% (in accordance with the percentage shown in Table A5).

[11] https://sites.google.com/site/santiagoavidal/projects/cover-methods
[12] http://www.sweethome3d.com

9. Threats to validity

The validity of our results depends on factors in the experimental settings. Next, we analyze four kinds of validity threats [6].

Conclusion validity: This threat concerns the statistical analysis of the results. Since we analyzed 30 Java applications we argue that the statistical power of the results is appropriate. However, we think that the analysis of library usage (Section 6) could be conducted with more applications to reduce the risk of drawing wrong conclusions.

Internal validity: This threat concerns causes that can affect the independent variable of the experiment without the researcher's knowledge. Our approach employs static analysis to identify over-exposed classes. For this reason, some limitations imposed by static analysis could bias our result. Specifically, some references to classes could not be identified by the static analysis algorithm of Moose (e.g. hierarchical relationships and reflective calls). This situation could lead to false positives due to missing calls or simply to the identification of classes that are not called.

Construct validity: It is concerned with the design of the experiment and the behavior of the subjects. Our main concern are those over-exposed classes that are intentionally left over-exposed by developers. While these classes are over-exposed they should not be labeled as such since the intention of the developers was to made them visible to other packages on purpose. However, we found that the number of public over-exposed classes being used by external applications is probably low (Section 6).

External validity: It is concerned with having a subject that is not representative of the population. We argue that the number of applications analyzed in the study is large enough to avoid this threat. Moreover, we distinguished and analyzed separately two kinds of applications: plain applications and libraries/frameworks.

10. Related work

Although other researchers have looked at class accessibility in object-oriented programming, they have not conducted a detailed empirical study as we report in this article. However, some works had analyzed the use of accessibility modifiers in classes, methods, and fields.

We previously presented an empirical evaluation of the over-exposed methods with the goal of analyzing their impact on information hiding and the interfaces of classes [4]. In this work we analyzed the same collection of applications which were also distinguished between plain applications and libraries/frameworks. However, our previous work was only focused on over-exposed methods while this work is focused on over-exposed classes. Also, similarly to this work, we previously analyzed the history of the applications with the goal of understanding the variations in the over-exposed methods. For these reasons, we think that the current article completes the empirical analysis of over-exposure.

Grothoff et al. [17] present a tool called JAMIT to restrict access modifiers. Differently from us, this work is not focused on modifiability but in security. Specifically, Grothoff et al. analyze whether a class is confined to the package to which it is declared. That is to say, they try to guarantee that a reference to a class cannot be obtained outside its package. This definition is stricter than ours, which has the modifiability goal of restricting the public API of classes/packages. In this context, Grothoff et al. use the JAMIT tool to report the percentage of classes that could be confinable. However, these values are not comparable with our over-exposed values because they are not exactly the same. A *confinable* class must satisfy some rules that an over-exposed class must not. For example, all methods invoked on a confined class must be *anonymous* [17], and all the subtypes of a confined class must be confined [17]. Moreover, differently from us, Grothoff et al. do not report the necessary refactoring to restrict class accessibility (e.g., change protected accessibility for default accessibility). Another important issue to mention is that their work do not distinguish between libraries/frameworks and plain applications as we do. In fact, they mix libraries and plain applications in their benchmark suite. Also, while Grothoff et al. consider nested classes, they do not distinguish between nested classes with different accessibilities as we do.

Bouillon et al. [3] present a tool that checks for over-exposed methods in Java applications. While it briefly mentions the classes with an accessibility broader than necessary, it does not make any analysis of them. Similar to ours, their tool determines the best access modifier by analyzing the references to each method. However, the tool was only tested in some packages of 4 applications (i.e. the applications were not carefully analyzed). The authors suggest that any over-exposed method could be the result of the developer's intention of extending the applications, but unlike our study in classes, no historical analysis is performed.

Müller [18] uses bytecode analysis to detect those access modifiers of methods and fields that should be more restrictive. However, the work does not describe the algorithm used to detect these situations nor presents case-studies to validate their tool.

Kobori et al. [19] investigated the evolution of over-exposed methods and fields for a set of open-source applications. Similarly to our analysis of classes, they reported that the change of access modifiers of methods is infrequent. They also found that the number of over-exposed methods and fields tends to increase in time.

Zoller and Schmolitzky [5] present a tool called AccessAnalysis to detect over-exposed methods and classes by analyzing the references to them. To measure the usage of access modifiers for types and methods, Zoller and Schmolitzky employ two software metrics: Inappropriate Generosity with Accessibility of Types (IGAT) and Inappropriate Generosity with Accessibility of Methods (IGAM). IGAT is equivalent to our concept of class over-exposure. To evaluate AccessAnalysis, the authors report on the analysis of 12 open-source applications. Their findings include that “general access modifiers are often chosen more generously than necessary”, which is in agreement with our observations. Specifically, Zoller and Schmolitzky reported results for 5 applications, which we also analyzed with our approach and showed very similar results. For example,

they reported an over-exposure of 20% for PMD while we reported 20.03%. However, Zoller and Schmolitzky do not analyze libraries/frameworks nor analyze the creation (and evolution) of the over-exposure phenomenon.

Steimann and Thies [20] highlight the difficulties of carrying out refactoring in the presence of non-public classes and methods. The authors formalize accessibility constraints in order to check the preconditions of a refactoring (e.g., moving a class to another package requires checking whether the accessibility of the class allows its users to still reference it). In particular, the authors analyze the cases in which a class or a method is moved between packages or classes with the goal of adapting their access modifiers to preserve the original behavior. They propose the change accessibility refactoring to change the access modifier of a declared entity. This refactoring recursively changes all the entities that are directly or indirectly related to the refactored entity. However, the approach presented by these authors does not analyze the detection of the over-exposure in a broad context.

11. Conclusions

Selecting the right accessibility modifiers of classes improves system modularity by avoiding unwanted interactions among components. We have proposed a code anomaly called *class over-exposure* to identify classes with an accessibility broader than necessary. We have empirically measured for a set of 30 applications that around 35% of plain Java public classes are over-exposed. Moreover, we found that systems define around 87% of their classes as public, with libraries defining more public classes than applications (RQ1). Also, we found that around 35% of the public classes are over-exposed (RQ2). Additionally, we analyzed the over-exposed classes of libraries and we determined that most of them are not used by third-party applications (RQ3). We also found that over-exposed classes often have this condition since its first implementation (RQ4). Finally, we could not find evidence that over-exposed classes are a major contributor to modularity problems (RQ5).

As future work we plan to:

- analyze the role of automated mechanisms of IDEs to generate source code in the creation of over-exposed classes;
- monitor programming activity to see how often class accessibility is re-considered by developers in the context of IDEs;
- analyze the effects of classes and methods over-exposed in the degradation of the software architecture (e.g., modularity, drift and erosion problems);
- extend our analysis of over-exposure to Java interfaces;
- refine our analysis by considering the use of reflection, increasing the number of analyzable classes.

Appendix A

See Fig. A1 and Tables A1–A6.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
a: ant														O				X	
b: check			X															O	
c: coveragedata		X		O	X								X	X	X		X	X	
d: coveragedata.countermaps			X											O					
e: instrument			X																
f: javancss							X	X	X	X	X			X	O		O		
g: javancss.ccl						X									O				
h: javancss.parser						X			O	O	O				O		O		
i: javancss.parser.debug						X		O		O	O								
j: javancss.parser.java15						X		O	O		O								
k: javancss.parser.java15.debug						X		O	O	O									
l: javancss.test																			
m: merge			X			O												O	
n: reporting			X			X									X		X	X	O
o: reporting.hlml			X			O	O	O						X		O		X	
p: reporting.html.files																O			
q: reporting.xml			X			O		O						X				X	
r: util	X	O	O										O	X	X		X		
s: webapp																			

Fig. A1. An average-case design structure matrix of Cobertura when considering over-exposed classes ('X' means existing dependencies and 'O' unnecessary class exposure).

Table A1
General information library/frameworks.

Library/framework	Version	#Classes	%PPC	%PDC	%NPC	%NDC	%NProC	%NPriC
Ant	1.8.3	1507	71.27	1.73	16.39	1.13	1.13	8.36
Commons-Compress	1.4.1	171	81.29	9.36	1.75	1.17	0	6.43
Commons-Primitives	1.0	433	66.05	22.63	0	6.47	3.23	1.62
Dom4J	2.0	158	78.48	10.13	0.63	0.63	4.43	5.69
Hibernate	4.1.3	5555	86.59	2.72	5.02	2.25	0.41	3.0
JavAssist	3.12	347	59.37	11.53	9.51	17.29	0.86	1.44
Jericho-HTML	3.2	164	39.02	35.37	0.61	5.49	0	19.51
JFreeChart	1.0.14	948	93.25	1.79	0.74	3.16	0.74	0.32
JHotDraw	7.0.6	309	85.11	2.27	3.88	0	0	8.74
JUnit	4.10	707	37.77	0.85	55.16	2.12	0	4.1
Log4J	1.2.16	435	71.49	10.11	3.68	3.22	0.23	11.26
Maven	3.0.4	732	82.1	7.79	0.96	3.42	0.41	5.33
Struts	2.3.3	2005	81.05	2.69	8.18	4.19	0.5	3.39
Tomcat	7.0.27	1951	65.91	4.31	10.81	1.33	6.25	11.38
Xalan	2.7.1	1091	64.89	19.62	6.32	5.13	0.27	3.76
Average		1101	70.91	9.53	8.24	3.8	1.23	6.29

PPC: plain public classes; PDC: plain default classes; NPC: nested public classes; NDC: nested default classes; NProC: nested protected classes; NPriC: nested private classes.

Table A2
Over-exposure information library/frameworks.

Library/framework	%OE Classes	%OE PPC	%OE NPC	%OE NDC	%OE NProC
Ant	27.87	16.39	9.62	0.8	1.06
Commons-Compress	23.39	21.05	1.75	0.58	0
Commons-Primitives	53.58	45.5	0	4.85	3.23
Dom4J	32.28	26.58	0.63	0.63	4.43
Hibernate	40.32	35.73	2.99	1.21	0.4
JavAssist	40.35	17.0	7.2	15.27	0.86
Jericho-HTML	10.98	6.71	0.61	3.66	0
JFreeChart	8.23	3.9	0.42	3.16	0.74
JHotDraw	34.3	31.07	3.24	0	0
JUnit	16.55	7.21	7.92	1.41	0
Log4J	25.52	19.31	2.76	3.22	0.23
Maven	18.99	14.89	0.55	3.28	0.27
Struts	33.12	21.8	7.03	3.79	0.5
Tomcat	38.54	22.14	9.02	1.13	6.25
Xalan	23.46	12.83	5.68	4.67	0.27
Average	28.5	20.14	3.96	3.18	1.22

OE PPC: over-exposed plain public classes; OE NPC: over-exposed nested public classes; OE NDC: over-exposed nested default classes; OE NProC: over-exposed nested protected classes.

Table A3
Incidence of JUnit tests in library/frameworks.

Library/framework	#PPCT	% PPCT
Ant	32	2.98
Commons-Compress	0	0
Commons-Primitives	50	17.48
Hibernate	313	6.51
Jericho-HTML	0	0
JFreeChart	166	18.78
JUnit	20	7.49
Log4J	6	1.93
Maven	32	5.32
Struts	114	7.02
Tomcat	20	1.56
Average		6.28

PPCT: plain public classes only externally referenced by tests.

Table A4
General information plain applications.

Application	Version	#Classes	%PPC	%PDC	%NPC	%NDC	%NProC	%NPriC
Argo	0.34	2087	63.82	22.23	0.43	3.45	0.96	9.1
Azureus	4.7.12	2797	81.27	1.07	3.93	1.57	6.65	5.51
Checkstyle	5.5	1029	66.47	17.88	3.89	5.54	0.78	5.44
Cobertura	1.9.4.1	148	83.11	3.38	4.73	1.35	0	7.43
FindBugs	2.0.1	1191	77.41	1.09	5.21	6.63	0.17	9.49
FreeMind	0.9	684	54.82	4.53	7.46	3.36	3.8	26.02
Jajuk	1.9.6	596	82.55	5.2	4.87	2.68	0	4.7
JEdit	5.0	1000	40.7	9.8	8	20.8	0.3	20.4
Jmol	12.2.33	706	59.77	17	1.98	16.57	1.98	2.69
Jstock	1.06	341	71.26	0	6.74	1.76	0	20.23
Logisim	0.0.1-a	1023	35.68	24.54	2.64	2.74	0	34.41
PMD	4.2.6	789	88.97	0.25	2.79	0.38	0.13	7.48
Portecle	1.7	98	42.86	35.71	0	1.02	0	20.41
SweetHome3D	3.5	396	46.21	4.3	2.78	0	1.52	45.2
TuxGuitar	1.2	782	89.13	0.9	0.77	0.13	2.17	6.91
Average		911.13	65.6	9.86	3.75	4.53	1.23	15.03

Table A5
Over-exposure information plain applications.

Application	%OE Classes	%OE PPC	%OE NPC	%OE NDC	%OE NProC
Argo	23.96	19.6	0.14	3.26	0.96
Azureus	33.75	23.99	1.93	1.25	6.58
Checkstyle	12.73	10.2	1.17	1.26	0.1
Cobertura	37.16	31.08	4.73	1.35	0
FindBugs	32.91	23.85	3.27	5.63	0.18
FreeMind	25.58	16.23	3.51	2.33	3.51
Jajuk	31.21	26.85	2.52	1.85	0
JEdit	35.7	13.0	5.5	16.9	0.3
Jmol	32.86	15.44	0.99	14.73	1.7
Jstock	34.31	29.33	3.52	1.47	0
Logisim	7.82	6.65	0.59	0.59	0
PMD	20.03	18.38	1.14	0.38	0.13
Portecle	8.16	8.16	0	0	0
SweetHome3D	16.92	13.89	1.77	0	1.26
TuxGuitar	28.52	26.09	0.26	0	2.17
Average	25.44	18.85	2.07	3.4	1.12

Table A6
Incidence of JUnit tests in applications.

Library/framework	#PPCT	% PPCT
Checkstyle	5	0.73
Cobertura	6	4.88
FindBugs	1	0.11
FreeMind	12	3.2
Jajuk	7	1.42
Jstock	0	0
PMD	47	6.7
SweetHome3D	7	3.83
Average		2.61

PPCT: plain public classes only externally referenced by tests.

References

[1] Parnas DL. On the criteria to be used in decomposing systems into modules. Commun ACM 1972;15(12):1053–8, http://dx.doi.org/10.1145/361598.361623.
[2] Scott ML. Programming language pragmatics. 3rd ed. Burlington: Academic Press; 2009.
[3] Bouillon P, Grokinsky E, Steimann F. Controlling accessibility in agile projects with the access modifier. In: Paige RF, Meyer B, editors. TOOLS (46). Lecture notes in business information processing, vol. 11. Springer, Berlin, Heidelberg; 2008. p. 41–59.
[4] Vidal SA, Bergel A, Marcos C, Díaz-Pace JA. Understanding and addressing exhibitionism in Java: empirical research about method accessibility. Emp Softw Eng 2016;21:483–516, http://dx.doi.org/10.1007/s10664-015-9365-9.
[5] Zoller C, Schmolitzky A. Measuring inappropriate generosity with access modifiers in Java systems. In: IWSM/Mensura; 2012. p. 43–52.
[6] Wohlin C, Runeson P, Hst M, Ohlsson MC, Regnell B. Experimentation in software engineering. Berlin, Heidelberg: Springer; 2012.
[7] Lanza M, Marinescu R. Object-oriented metrics in practice – using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Berlin, Heidelberg: Springer; 2006.
[8] Sangal N, Jordan E, Sinha V, Jackson D. Using dependency models to manage complex software architecture. In: ACM sigplan notices, vol. 40. ACM, New York; 2005. p. 167–76.
[9] MacCormack A, Rusnak J, Baldwin CY. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. Manag Sci 2006;52(7):1015–30.
[10] Milev R, Muegge S, Weiss M. Design evolution of an open source project using an improved modularity metric. In: Open source ecosystems: diverse communities interacting. Springer, Berlin, Heidelberg; 2009. p. 20–33.
[11] Robbes R, Lungu M, Röthlisberger D. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering. ACM, New York; 2012. p. 56.
[12] Raemaekers S, Van Deursen A, Visser J. Semantic versioning versus breaking changes: a study of the maven repository. In: 2014 IEEE 14th international working conference on source code analysis and manipulation (SCAM). IEEE, Victoria, BC; 2014. p. 215–24.
[13] Booch G. Object-oriented analysis and design with applications, 2nd ed. Benjamin/Cummings series in object-oriented software engineering. Addison-Wesley, New York; 1995.
[14] Demeyer S, Ducasse S, Nierstrasz O. Object-oriented reengineering patterns. Bern: Morgan Kaufmann; 2003.
[15] Gamma E, Helm R, Johnson RE. Design patterns. Elements of reusable object-oriented software. 1st ed.Addison-Wesley Longman, New York; 1995.
[16] Mens T, Demeyer S. Software evolution. Springer, Berlin; 2008, iSBN 978-3-540-76439-7, http://dx.doi.org/10.1007/978-3-540-76440-3.
[17] Grothoff C, Palsberg J, Vitek J. Encapsulating objects with confined types. ACM Trans Program Lang Syst (TOPLAS) 2007;29(6):32.
[18] Müller A. Bytecode analysis for checking Java access modifiers. In: Work in progress and poster session, 8th international conference on principles and practice of programming in Java (PPPJ 2010), Vienna, Austria; 2010.
[19] Kobori K, Matsushita M, Inoue K. Evolution analysis for accessibility excessiveness in Java. In: 2015 IEEE 22nd international conference on software analysis, evolution and reengineering (SANER). IEEE, New York; 2015. p. 83–90.
[20] Steimann F, Thies A. From public to private to absent: refactoring Java programs under constrained accessibility. In: Drossopoulou S, editor. ECOOP. Lecture notes in computer science, vol. 5653. Springer, Berlin; 2009. p. 419–43.