Noname manuscript No.
(will be inserted by the editor)

Understanding and Addressing Exhibitionism in Java
Empirical Research about Method Accessibility

Santiago A. Vidal · Alexandre Bergel · Claudia Marcos · J. Andrés Díaz-Pace

Received: date / Accepted: date

Abstract Information hiding is a positive consequence of properly defining component interfaces. Unfortunately, determining what should constitute a public interface remains difficult. We have analyzed over 3.6 million lines of Java open-source code and found that on the average, at least 20% of defined methods are over-exposed, thus threatening public interfaces to unnecessary exposure.

Such over-exposed methods may have their accessibility reduced to exactly reflect the method usage. We have identified three patterns in the source code to identify over-exposed methods. We also propose an Eclipse plugin to guide practitioners in identifying over-exposed methods and refactoring their applications. Our plugin has been successfully used to refactor a non-trivial application.

Keywords Method accessibility · Information hiding

1 Introduction

Developing activities are centered on the premise that software is made to be changed (Booch 2004). Limiting the impact of a component modification to the rest of the system is known to be particularly difficult (Robbes et al 2012). It is widely recognized that encapsulation and information hiding play a key role in software maintenance and evolution. In his seminal contribution (Parnas 1972), David L. Parnas phrased:

Santiago A. Vidal
ISISTAN, UNICEN, Argentina and CONICET
E-mail: svidal@exa.unicen.edu.ar

Alexandre Bergel
Pleiad Lab, Department of Computer Science (DCC), University of Chile
E-mail: abergel@dcc.uchile.cl

Claudia Marcos
ISISTAN, UNICEN, Argentina and CIC
E-mail: cmarcos@exa.unicen.edu.ar

J. Andrés Díaz-Pace
ISISTAN, UNICEN, Argentina and CONICET
E-mail: adiaz@exa.unicen.edu.ar

> *"Every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition is chosen to reveal as little as possible about its inner workings."*

Parnas' work closely associates the notion of information hiding with component interfaces. Among the different interfaces a software component may have (Meyer 2009), its *public interface* determines which services may be used and by whom. Reducing the public interface to a minimum is an elementary design rule in software engineering that has received scarce attention from the research community (Riel 1996; Martin 2002; Zoller and Schmolitzky 2012; Steimann and Thies 2009).

Method accessibility has a direct impact on the public interface by enforcing information hiding, one of the key features of object-oriented programming. Over-exposed methods are associated with a strong negative aspect: a programmer may wrongly consider an over-exposed method as part of the public interface. Each public method is an entry point for the class itself and the web of classes connected to that class, thus the more entry points a class has, the more likely the runtime state of an object is to be exposed.

Despite the advances in programming environments and methodologies, little assistance is offered to programmers to define the public interface of classes. Along this line, we think that the developer should be assisted in this task.

Our analysis of 30 open-source Java applications reveals that at least 20% of the methods are defined with an accessibility that is broader than necessary: a typical situation is when a method is declared as public, whereas it may simply be protected or private. Consider the following situation inspired by one of our case studies:

```
public class Author {
  private String name;

  public Author (String name) { giveName(name); }

  // giveName is never called outside Author
  public void giveName (String aName) { this.name = aName; }

  public String getName () { return name; }
}
```

The class `Author` defines a constructor and two public methods. Being public allows any other method in the system to merely invoke `giveName(...)` and `getName ()`. The constructor `Author` calls method `giveName(...)` to set a name. In the *whole application*, method `giveName(...)` is referenced and called nowhere, except by the class constructor. Method `giveName()` may therefore be private without affecting the application integrity. The programmer probably made `giveName(...)` public on the assumption that setting an author name is an operation important enough to be used by client classes, either in future versions of the application or in external components. However, in the current version of the system, this assumption is a mere speculation because there is no evidence that this method is useful outside class `Author`. An *over-exposed* method is a method with an accessibility broader than necessary based on the location of its caller methods, as the method `giveName(...)` in the example given above. Note that a method can be over-exposed as part of the developer's design intent, either to support future application evolution or to usage

by external components, but it can also be over-exposed because of "over-design" or by mistake.

Mainstream programming languages have a sophisticated access modifiers system for its methods and classes. Unfortunately no assistance is offered to a programmer to properly pick the right accessibility. This article contributes to rectifying this situation by carefully answering relevant questions and providing a robust prototype.

To understand the extent of the over-exposure phenomenon, we have studied over 3.6 millions lines of Java code, looking at how method accessibility manifests in practice. We structure our empirical analysis along the following research questions:

- Q1 - *Is there a difference in terms of method accessibility distribution between libraries / frameworks and plain applications?*
- Q2 - *Do libraries and frameworks contain, on average, more over-exposed methods than plain applications?*
- Q3 - *Are over-exposed methods effectively used in future system versions?*

In order to answer these questions, we provide three code patterns that represent situations where a method is over-exposed. These patterns are based on a combination of invocations between methods and classes. Using these patterns as detectors, over-exposed methods may then be refactored to reduce their accessibility to their strict necessity. The size of the public interface of classes will be consequently reduced.

Additionally, to assist the refactoring of method accessibility, we have developed an Eclipse plugin to automatically identify over-exposed methods and propose refactorings to remove the unnecessary method exposure.

To verify that no changes are observed in the behavior of applications when reducing method accessibility, we refactored SweetHome3D, a 84K LOC Java application. Based on a series of tests, no impact on its behavior at runtime has been observed. This gives us confidence that the semantics of the application are preserved to some extent, after reducing the accessibility.

This article makes the following contributions:

- It highlights a limitation of programming environments to assist programmers in rightfully choosing the access modifier of a method.
- It empirically studies the presence of over-exposed methods which has not been rigorously been covered in the past.
- It provides three code patterns to identify over-exposed methods.
- It describes the implementation and the evaluation of a prototype to efficiently identify over-exposed methods.

Outline. The article is structured as follows. Section 2 briefly summarizes the different access modifiers for a method in Java. The section further analyzes a set of 15 libraries and 15 plain applications. Section 3 discusses three code patterns for detecting over-exposed methods, and then analyzes the proportion of exposure in our 30 applications. Section 4 monitors changes in over-exposed methods over application versions. Section 5 discusses possible reasons for finding so many over-exposed methods. Section 6 presents the threats of validity of our approach. Section 7 briefly presents and evaluates our Eclipse plugin for identifying and refactoring over-exposed methods. Section 8 describes a case-study conducted on a

Accessibility	class	package	subclass	world
`public`	A	A	A	A
`protected`	A	A	A	-
`package`	A	A	-	-
`private`	A	-	-	-

Fig. 1 Method accessibility in Java (A = accessible, - = not accessible).

non-trivial Java application. Section 9 analyzes related work. Section 10 concludes and discusses future lines of work.

2 Method Accessibility in Java Applications

This section discusses the access modifiers offered by the Java programming language and their presence in a set of 30 Java applications.

2.1 Access modifiers offered by Java

The Java programming language gives to each field and method one of four different accessibilities. The accessibility of a method m unambiguously determines which methods in the system have the right to invoke m. This "right" depends on the class and the package of the calling method.

A *public* method may be invoked by any method. A *protected* method may be invoked only by (i) the classes that belong to the same package of the protected method and (ii) the subclasses of the class that defines the protected method. A *package* method may be invoked only by the methods of the same package. Package is the accessibility per default, when no accessibility is specified (*i.e.,* method declared without an access modifier). A *private* method may be invoked only by its defining class.

These four accessibilities may be ordered along the degree of exposure a method may have (cf Figure 1, inspired by a Java tutorial[1]). A public method is more exposed than a protected method, itself more exposed than a package or a private method.

The Java compiler makes sure that each method call conforms to the accessibility of the targeted method. A compilation error is reported if the targeted method is not accessible for a caller. On the contrary, no error or warning is provided if a method is accessible to more methods than necessary. The premise on which this paper is based is that the more public methods a class has, the more exposed it is. On the opposite, the more private methods a class has, the less exposed it is.

We informally define a class C as "exhibitionist" if parts of C are unnecessarily accessible to other classes via methods.

2.2 Study of Java Applications

We have selected 30 open-source Java applications and measured the use of access modifiers for methods. The appendix lists these applications and the results of the

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

metrics relevant to our study in tables 4, 5, 6 and 7. Some of these applications were chosen because of their popularity others were found on the website sourceforge.net, a popular hosting platform for open source software projects. In all the cases we checked for the availability of its source code and a strong commitment of their supporting community (*e.g.,* existence of an active mailing list, availability of unit tests). Additionally, we sought to select applications involving different sizes and belonging to different domains. The size of the applications we are considering ranges from 15K to 689K lines of code. We have analyzed nearly 275,000 methods totaling over 3.6 million lines of code. The appendix contains all the software versions to let one easily reproduce our findings. Additionally, the datasets[2] and the processing code[3] used to conduct the experiments are available for download.

In this paper we focus on method accessibility but we do not consider variable accessibility. While the analysis of variable accessibility is relevant, we have focused on methods because their accessibility is essential to the notion of encapsulation and it is a topic largely under-considered by both practitioners and researchers.

We classify these applications into two distinct categories:

- *library / framework* - applications that are either self-claimed as a library (*e.g.,* JUnit, Struts) or applications that are meant to complement functionalities offered by Java (*e.g.,* Commons-Primitives, Commons-Compress). These applications are meant to be used by other applications, and cannot be directly considered as an end product for a non-programmer.
- *plain application* - applications that are meant by their authors to be used directly by an end-user (*e.g.,* Jedit, Jmol). We include in this category applications that operate on other applications (*e.g.,* PMD, Cobertura, FindBugs). Although such applications are also meant to be extended, we the end user the primary user of these applications, with extensions meant as possibilities to add features for these users.

The reason for these two categories stems from the two very different ways of using these applications. A library is meant to be extended and/or used by an application. A library has therefore to provide public services and hooks in the source code to be easily extended. A plain application does not necessarily have the same constraints, since its typical usage does not involve an extension of its source code.

We therefore hypothesize that *a library has more public methods and fewer private methods than an application for end users.* We further expect to find a larger number of methods without calls in library systems than in plain applications, since a library defines methods that will be called when the library is used by third applications.

2.3 Accessibility in Java applications

We have studied the distribution of method accessibilities across libraries and plain applications. This section addresses both: the hypothesis presented in the previous section and the $Q1$ research question stated earlier. Specifically, in this section we try to determine if there is any statistical significant difference in the number of methods defined with a given accessibility between plain application and libraries.

[2] http://bit.ly/mseFiles

[3] http://ss3.gemstone.com/ss/SPIRIT.html (package Spirit-ExhibitionismTests)

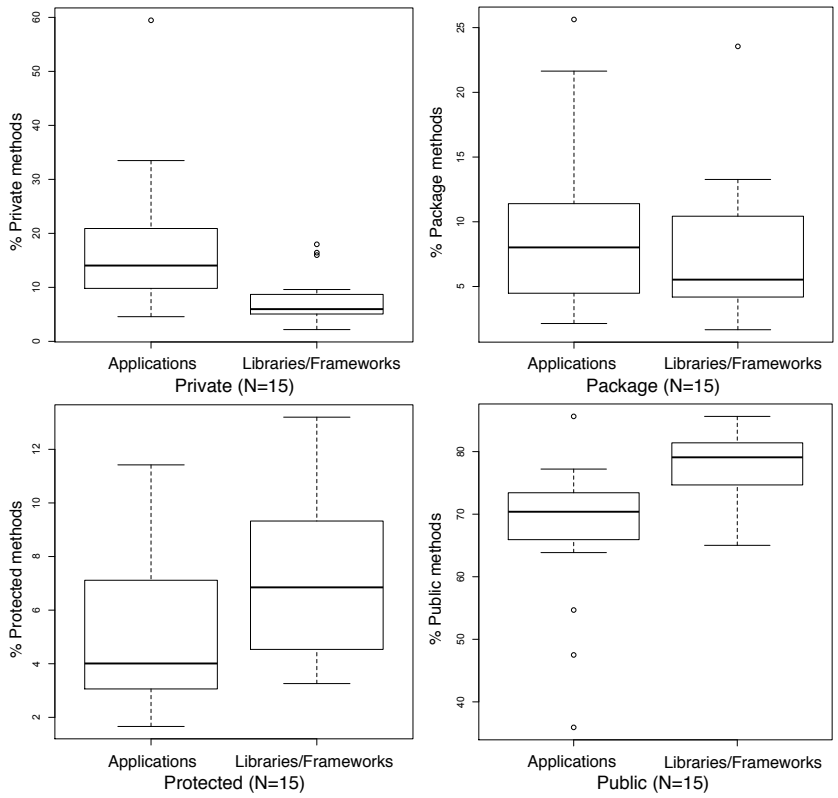

Fig. 2 Accessibility distribution

Figure 2 gives the distribution of the four access modifiers for methods (public, private, protected, and package) for the 30 Java applications of our study. Each chart is a box plot, plotting the frequency against the number of methods of a particular accessibility. For example, the top left chart gives the number of applications with a particular portion range of private methods. The metrics we obtained from the applications have a precision of 0.01% (cf Appendix), meaning that we rounded up the values to the second decimal place.

Public methods. On average, plain applications have 67.1% of public methods. The standard deviation is 12.5%. Libraries have an average of 77.2% of public methods, with a standard deviation of 6.5%. That is, libraries report on average more public methods than plain applications.

Although the box plots give descriptive insights, no conclusion can be made so far on whether there is a significant difference between the distribution of public methods between libraries and plain applications. A statistical test is necessary.

First, we test the data for normality using the Shapiro-Wilks test. Table 1 shows the p-values obtained from the tests for the different accessibility data. Since

Accessibility modifier	p-value
Public	0.001708
Protected	0.04801
Package	0.002281
Private	0.00511

Table 1 Shapiro-Wilks test results (normality).

Factor	n	T^+	T^-
Public	15	82	38
Protected	15	79	41
Package	15	26	94
Private	15	30	90

Table 2 Mann-Whitney-test results.

all the p-values are lower than 0.05, we can conclude that the data deviate from normality.

As the data of public method accessibilities is not normally distributed across applications, we use the Mann-Whitney-test to check the difference between plain applications and libraries on defining public methods. In this context, we define the following null hypothesis:

– H_0: there is no difference between plain applications and libraries on how frequently public accessibility is used

The alternative hypothesis (H_1) is that there is a difference between plain applications and libraries with regards to how often the public accessibility is used. Table 2 describes the T^+ and T^- values which are calculated based on the sum of the ranks between the percentage of methods with an accessibility modifier. The H_0 hypothesis could be rejected if the smallest of T^+ and T^- is less than or equal to a value indicated in a statistical table (Wohlin et al 2000). In this case, for $n = 15$ the value is 25. From Table 2, the null hypothesis H_0 cannot be rejected with a two-tailed test with a probability of error (or significance level) α= 0.05 (i.e. there is a 5% chance of wrongly accepted H_0) and a p-value of 0.007544 since min(T^+, T^-) is larger than 25 (Siegel and Castellan 1988). That is, *there is no statistically significant difference between plain applications and libraries on how public accessibility is used.*

As we have just seen, a large proportion of methods are declared public in Java applications (around 70%). This is a rather surprising fact since information hiding and encapsulation principles promote the idea of restricting public interfaces to their minimum. That is, the accessibility of the methods should be reduced to their strict necessity based on the locations of caller methods. Our analysis shows the opposite tendency in practice.

Protected methods. Protected methods are used much less than the other accessibilities. Plain applications have on average 5.1% (standard deviation = 2.9) protected methods and libraries have 7.2% (standard deviation = 3.4). Similarly to public methods, we define the null hypothesis as:

– H_0: there is no difference between plain applications and libraries on how frequently protected accessibility is used

As shown in Table 2, a new Mann-Whitney-test for these measurements failed to reject the null hypothesis with a significance level of 0.05 and a p-value of 0.0742 since min(T^+, T^-)=41.

Package methods. Plain applications have an average of 9.8% (standard deviation = 7.0) package accessible methods and libraries have an average of 7.6% (standard deviation = 5.7). The null hypothesis for the Mann-Whitney-test is formulated similarly to that of the previous accessibilities. From Table 2, the null hypothesis cannot be rejected with a two-tailed test with a probability of error of 0.05 and a p-value of 0.6832 since min(T^+, T^-)=26.

Private methods. Plain applications have an average of 18% of private methods (standard deviation = 13.8) and libraries have an average of 7.9% (standard deviation = 5). The null hypothesis for the Mann-Whitney-test is formulated similarly to the previous accessibilities. The test returns a p-value of 0.002306 with a probability of error of 0.05 and min(T^+, T^-)=30, which is comparable to the result of the previous analyse. That is to say, there is not enough statistical evidence to reject the null hypothesis. That means that *there is no statistically significant difference between plain applications and libraries on how private accessibility is used.* Surprisingly, we found that there are about 2.2 times more private methods in plain applications than in libraries.

Note that the application Cobertura is an outlier, as shown in Table 6 of the appendix. This 50K LOC application is composed of 3,313 methods, in which 1,970 are private and 1,190 are public. Cobertura has 59.46% private methods, which is a higher number than for the remaining applications. A closer look at this application reveals that a large portion of these private methods belong to a built-in Java parser. This parser has been automatically generated by the JavaCC parser generator.

In summary, answering Q1 showed that there is not a difference in terms of method accessibility distribution between libraries/frameworks and plain applications. Also, we have found that, on average, 70% of the methods are defined as public. All the experiments given in the article have been performed on the Moose software analysis platform[4]. Moose offers a meta-model on which we formulate queries and compute metrics. The parsing of the Java application has been done using the VerveineJ Java analyzer[5].

3 Over-exposed Methods

We qualify *a method as over-exposed if it has an accessibility that is greater than the one being necessary.* Necessity here should be interpreted in the "context" of the application, which can be that of a plain application or a library. As a consequence, an over-exposed method may have its accessibility reduced to reflect its actual use. Being over-exposed for a method depends on (i) other methods that call the

[4] http://www.moosetechnology.org
[5] http://www.moosetechnology.org/tools/verveinej

over-exposed method and (ii) the accessibility of the original method in presence of overriding.

A typical scenario for a method to be over-exposed is when the method is declared public and used as if it were a private method (*i.e.,* solely called within its class). This method is over-exposed and its accessibility could be restricted without impacting the application behavior. An example of such a situation is the class `Author` given earlier (Section 1).

3.1 Accessibility patterns

In this section, we present an approach to identify over-exposed methods through three code patterns. We illustrate these patterns using contrived but representative examples.

Pattern 1 - Package method. A method *a()* defined as *public* or *protected* in a class C is over-exposed and its accessibility may be changed to *package* if:

- it is called by at least one method that is not defined in C and,
- all the caller methods are defined in the same package as *a()*

Example: Classes `Library` and `Author` live in the same package:

```
package library;
public class Library {
  public void defineNewAuthor() {
   new Author().giveName();
  }
}

package library;
public class Author {
  public void giveName() { ... }
}
```

Method `Library.defineNewAuthor()` invokes `Author.giveName()`. Method `giveName()` is not called anywhere else except by `Library`. The most restricted accessibility allowed for `giveName()` is package. If `giveName()` is protected or public, then it is over-exposed.

Pattern 2 - Protected method. A method defined as *public* in a class C is over-exposed and its accessibility may be changed to *protected* if:

- it is only called by methods defined in classes that inherit from C

Example: `IndexedAuthor` and `Author` are two classes living in different packages:

```
package library;
public class Author {
  public void giveName() { ... }
}

package indexedlibrary;
public class IndexedAuthor extends library.Author {
  public void computeIndex() { this.giveName(); }
}
```

`IndexedAuthor` invokes `giveName()`, which is defined in a superclass. Method `giveName()` is not called anywhere except by `IndexedAuthor`. The accessibility of `giveName()` is public. The minimum accessibility for method `giveName()` is protected, and it is over-exposed if it is public.

Pattern 3 - Private method. A method *a()* defined as *public*, *package* or *protected* in a class C is over-exposed and its accessibility may be changed to *private* if:

– it is only called by methods defined in *Foo*

Example: Class `Author` defines `giveName()`, which is declared as public:

```
package library;
public class Library {
  public void defineNewAuthor() {
   new Author().defineNewName();
  }
}

package library;
public class Author {
  public void giveName() { ... }
  public void defineNewName() { this.giveName(); }
}
```

Method `giveName()` is solely called by `Author` itself. If the accessibility of `giveName()` is either public, protected or package, then it is over-exposed. The minimum accessibility for method `giveName()` that is strictly necessary is private. Note that `giveName()` does not override any method. Because the accessibility of an overridden method cannot be more restrictive than that of the method that it is overriding (this case is discussed in Section 3.3), overridden methods are considered in our patterns only when their accessibilities can be effectively reduced. If an overridden method fits with one of the patterns but its accessibility cannot be reduced because of the method that is being overridden accessibility, the method is not considered over-exposed.

To sum up, we qualify a method as over-exposed (*e.g.,* `giveName()`) if it is involved in at least one of the three patterns described above. Note that a private method cannot be over-exposed since private is the most restrictive accessibility.

3.2 Proportion of unnecessary exposure

We have seen that libraries and plain applications have a slightly different profile of method accessibility (Section 2.3). This finding therefore suggests that these two kinds of applications should be treated distinctly. This section analyzes the distribution of the unnecessary method exposure of 30 Java applications. The source code of the 30 applications (15 plain applications and 15 libraries) we analyzed totals 457,351 method invocations.

Libraries and frameworks. Not all the methods defined in an application are relevant for our analysis. We consider a method m analyzable if (i) m is called at least once by another method and (ii) m is non-private. In order to determine whether the accessibility of m is appropriate, m must be called. If this is not the case,

then it cannot be involved in one of the patterns P1, P2, or P3, presented earlier (Section 3.1). Note that methods that are not analyzable in our study might or might not be over-exposed.

For the total number of methods, the range of over-exposed methods goes from 11.15% to 32.98%, with an average of 24.81%, a median of 26.41%, and a standard deviation is 5.93. For calculating these values, we took into account the calls from the test methods of the libraries.

The 15 libraries define 138,568 methods, for which 8,541 are private. The following table gives the proportion of the accessibility for non-private methods:

methods	*defined*	*analyzable*	*right accessibility*	*over exposed*
public	110,438	44,771	17,056	27,715
protected	9,220	5,826	2,145	3,681
package	10,369	2,435	1,834	601
total	130,027	53,032	21,035	31,997

The *defined* column gives the number of methods that are defined. In total we have 130,027 non-private methods (138,568 - 8,541). Note that we also include abstract methods, since an abstract method has an access modifier making it relevant to our analysis (one of its implementations may be invoked). The *analyzable* column gives the number of methods being called at least once for each kind of accessibility, and finally it totals the number of methods being called at least once that are not private (53,032). The *right accessibility* is the number of methods that have an adequate accessibility (*i.e.,* the accessibility that is strictly necessary), meaning that they do not fit into any of the three patterns presented earlier.

From the 130,027 methods, only 40.78% (= 53,032) are analyzable. The remaining 59.22% of methods are either not called by the actual applications or references to them could not be determined. Out of the 53,032 methods, 21,035 have an adequate accessibility, meaning that 31,997 are over-exposed. **In total, 23.09% of the 138,568 methods are over-exposed**.We found that 86.6% of the over-exposed methods are public methods whose accessibilities can be reduced. The following table details the changes suggested by our pattern-based analysis of the accessibility modifiers.

Current accessibility	Suggested accessibility	# of methods over-exposed
Public	Private	6798
Public	Protected	2612
Public	Package	18305
Protected	Private	3681
Package	Private	477
Package	Protected	124

Plain applications. Similar to the case of libraries, public methods make up a large proportion when compared to the other methods. The 15 plain applications define 144,795 methods, which include 16,017 private methods. We present the proportion of the accessibility for the 128,778 non-private methods:

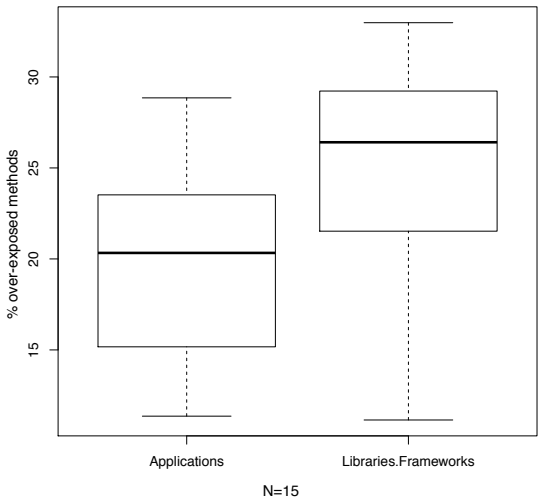

Fig. 3 Distribution of over-exposed methods in plain applications and libraries/frameworks

methods	*defined*	*analyzable*	*right accessibility*	*over exposed*
public	95,433	46,336	24,861	21,475
protected	9,744	7,192	4,423	2,769
package	14,551	4,429	3,617	812
total	128,778	57,957	32,901	25,056

Out of the 57,957 methods that are analyzable, 32,901 methods have an adequate accessibility. **As a consequence, 17.30% (= 25,056) of the 144,795 methods are over-exposed**. A 85.7% of the over-exposed methods are public methods whose accessibility should be reduced. The following table details the reductions suggested by our analysis.

Current accessibility	Suggested accessibility	# of methods over-exposed
Public	Private	5962
Public	Protected	1838
Public	Package	13675
Protected	Private	2769
Package	Private	702
Package	Protected	110

Libraries/frameworks versus plain applications. The libraries we analyzed have on the average 5.79% (= 23.09 - 17.30) more over-exposed methods than the plain applications do.

Figure 3 shows the distributions of over-exposed methods for libraries and plain applications. We tested the normality of the data using the Shapiro-Wilks test. We obtained a p-value of 0.2681. Since the p-value is higher than 0.05 we can conclude that the data is normal. The graph in Figure 3 follows the intuition that

libraries offer public services that are meant to be used by external applications. This analysis answers the second research question stated earlier ($Q2$, Section 1). Since the data is normal we can use a Student's t-test to check for any statistical difference between the number of over-exposed methods in libraries and plain applications. H_0 is that libraries have the same percentage of over-exposed methods than applications have. After testing we reject the null hypothesis with $\alpha = 0.05$ and p-value = 0.01 indicating that the two distributions are statistically different. That means that libraries have on average more over-exposed methods than plain applications.

Use of libraries / frameworks. We have earlier determined that 23.09% of the methods contained in libraries are over-exposed, a higher value than for plain applications (17.30%). This is not really surprising since the intention of a library is to be used or instantiated.The question that naturally follows is whether the number of over-exposed methods for libraries is reduced when they are used by external applications. Answering this question implies analyzing a number of client applications for each library. This is a significant amount of work that we leave for future efforts. To get a sense of the possible answer, we analyzed three applications that use the JFreeChart library. We measured the number of over-exposed methods in JFreeChart when used by each of the client applications. We then compared the number of over-exposed methods to that of JFreeChart alone.

Specifically, we analyzed the way in which iTracker[6], OpenReports[7] and JSky[8] use JFreeChart. We conducted an experiment similar to the one conducted with the libraries, but instead of focusing on the analysis of the methods of the client applications (*i.e.,* iTracker, OpenReports and JSky), we computed the number of over-exposed methods in the version of JFreeChart used by the applications. The following table compares our results for JFreeChart and the usage of this library in the three aforementioned applications:

system	*defined*	*analyzable*	*right accessibility*	*over exposed*
JFreeChart	8,207	3,456	985	2,471
iTracker	8,207	3,456	985	2,471
JSky	8,207	3,482	1,032	2,450
OpenReports	8,207	3,473	1,021	2,452

Without being used by a client application, JFreeChart defines 8,207 methods, from which 2,471 (= 30.1%) are over-exposed. When JFreeChart is used, the number of analyzable methods (*i.e.,* non-private methods that are called at least once) is slightly higher. These methods that turn into being analyzable are entry points of the library. The number of over-exposed methods is slightly reduced to 29.8%., presumably because some methods are effectively used by external clients (note that the values reported for iTracker are the same as the ones reported to JFreeChart because iTracker calls a subset of the methods called internally by JFreeChart). Still, this small analysis indicates that using a library may not strongly reduce the number of over-exposed methods. Nonetheless, more experiments are needed to confirm this claim.

[6] http://www.itracker.org
[7] http://oreports.com/
[8] http://archive.eso.org/cms/tools-documentation/jsky/

3.3 Discussion of the study

A number of points related to our measurements are worth discussing.

Callbacks. It is common to have methods defined in an application that are not directly called by the application itself. Methods intended to be called by the Java runtime, such as event callbacks, belong to this category. Since a method defining the callback is not directly called by the application, the method cannot be analyzed. Consider the following code excerpt found in SweetHome3D:

```
private void displayHome(final JRootPane rootPane, final Home home,
                final UserPreferences preferences,
                final ViewFactory viewFactory) {
    EventQueue.invokeLater(new Runnable() {
     public void run() {
        HomeController3D controller =
         new HomeController3D(home, preferences, viewFactory, null, null);
      rootPane.setContentPane((JComponent)controller.getView());
      rootPane.revalidate();
     }
    });
   }
```

Method `invokeLater(...)` takes as argument an instance of an anonymous class that implements the interface `Runnable`. This anonymous class implements the method `public void run()`. This method is invoked by a particular thread, called the dispatch thread by the Java runtime. The call of `run()` is therefore made in classes that belong to Java.

Our analysis focuses on what directly constitutes the applications, that is, their source code. We did not consider the runtime environment in order to avoid having redundancy for each analyzed application.

Use of reflection. We assessed the 30 applications by analyzing their source code. We therefore discarded all aspects that may occur at runtime. In addition to callbacks, one limitation of our approach is that it does not take into account reflective method invocations. As an example, consider the following code excerpt obtained from SweetHome3D:

```
public void destroy() {
    if (notNull(this.appletApplication)) {
      try {
        Method destroyMethod = this.appletApplication.getClass().getMethod("destroy", new
       Class [0]);
        destroyMethod.invoke(this.appletApplication, new Object [0]);
      } catch (Exception ex) {
        ex.printStackTrace();
      }
    }
    this.appletApplication = null;
    System.gc();
  }
```

An instance of the Java class `Method` is obtained, and then it is invoked using `destroyMethod.invoke(...)`. The call to the actual `destroy()` method cannot be statically identified. Although this method is called by the application itself, this call is not considered in our analysis.

In the 30 applications we analyzed, 23 make use of the reflective capabilities of Java. Of these 23, 16 are actually invoking methods via reflection. We analyzed the percentages of classes that invoke methods via reflection. On average we found that only the 2.98% of the classes of the libraries and 0.7% of the classes of the applications use reflection. While the values obtained for the different applications were similar, there were two outliers in the libraries: JavaAssist (9.24%) and JUnit (7.3%). We think that these values are related to the domain of the libraries (bytecode manipulation and testing, respectively). We did not specifically measure the impact of reflection in our analysis. We plan to investigate this aspect in the future using a dynamic analysis, similarly to Thies and Bodden (2012).

Inheritance and accessibility. Java allows overriding methods to have their accessibility widened. It means that (i) a package method may be made protected or public and (ii) a protected method may be made public when being overridden[9]. Consider the following example:

```
public class Author {
  void giveName() { ...}
}

public class IndexedAuthor extends Author {
  public void giveName() { ...}
}

public class KeyedAuthor extends Author {
  protected void giveName() { ...}
}
```

Each override has an accessibility wider than the original method of the root class. The class `Author` defines the method `giveName()`, which has a package accessibility. `IndexedAuthor` redefines it and makes it public. `KeyedAuthor` redefines the `giveName()` method as protected.

Among the 283,363 methods we analyzed, we found only 1,339 (= 0.47%) occurrences of overriding methods that have a wider accessibility than the declaration in a superclass.

4 Monitoring the Evolution

This section assesses whether the presence of over-exposed methods is intended to satisfy future needs of different applications. The reported results address our third research question ($Q3$). For each of the plain applications, we analyzed and compared the evolution of over-exposed methods along the history of the applications. We could not find more than one version of the Logisim application and therefore report our measurements for the 14 remaining applications. We have analyzed 7 versions for all but one application. Only 5 versions are publicly available for CheckStyle. For each application A, we denote A_x the version x of A. The argument x ranges from 0 to 6.

[9] Note that a private method cannot be overridden.

4.1 Evolution of over-exposed methods

For each version of each application, we measured the number of defined methods, over-exposed methods, and the ratio between these two. Our results are presented in Figure 7, given in the appendix. Each application comes with two graphs:

- the graph with two curves located on the left-hand side shows the evolution of the total number of methods with the number of over-exposed methods.
- the graph with one curve located on the right-hand side indicates the evolution of the relative number of over-exposed methods.

These graphs visually convey the intuition that the number of over-exposed methods seems to correlate with the total number of methods. In fact, plotting all the pairs *(# methods in A_x, # over-exposed methods in A_x)* indicates a linear correlation between these two[10]. We therefore computed Spearman's correlation coefficient (denoted p). We found that 10 of the 14 applications[11] have a strong positive correlation (> 0.8) between the number of defined methods and the number of over-exposed methods. Although we cannot deduce the causality between these two, the strong correlation indicates that a new application version that contains more methods is likely to have more over-exposed methods than in its previous version.

Out of the remaining five applications, we distinguish three applications with little variation in their number of defined methods and over-exposed methods. Checkstyle ($r = 0.23$), Jajuk ($r = 0.82$) and Jedit ($r = 0.18$) loosely correlate because of the small variations in their corresponding measurements. The two remaining applications, Jmol ($r = 0.04$) and Portecle ($r = -0.90$), went through some major change, which breaks the continuity of our measurements, thus resulting in a low correlation.

4.2 What do over-exposed methods become?

One question that naturally arises is what do over-exposed method become over time. We provide an answer to this question by monitoring each over-exposed method of our applications over time. There are three different fates for a method that is over-exposed. A method that is over-exposed in a version A_x, may in a version A_y ($y > x$):

- *be not over-exposed anymore* – This happens if the method has new calling methods that fit well with its accessibility. Based on the `Author` class given in the introduction, the method `giveName(...)` may be called from another package in version y of the application that contains `Author`.
- *not exist anymore* – This situation corresponds to a method removal or renaming. Version y does not contain the method `Author.giveName(...)`.
- *remain over-exposed* – The method is still over-exposed in version y. This does not prevent the method from having additional calling methods, however its accessibility remains still too permissive.

[10] These scatterplots are not reported in this paper.

[11] ArgoUML, SweetHome3D, FreeMind, Ant, Cobertura, Findbugs, Jajuk, JStock, PMD, TuxGuitar

We measured the proportion of over-exposed methods that fall into each of these three situations by tracing each over-exposed method found in an early version of each application. For each application A, we compare the over-exposed methods found in A_0 (the initial version) with the methods found in a later version; A_0 is therefore used as a reference point. The six last versions are denoted $A_1, ..., A_6$, from the oldest to the newest one. Figure 8 shows four metrics for each application:

- *Over-exposed methods (OEM) in* A_0: this value simply corresponds to the number of over-exposed methods in A_0. This value is equal to the sum of the following three metrics.
- *OEM in* A_0 *that are* ***not*** *OEM in* A_x: The number of over-exposed methods in A_0 that are not over-exposed in A_x.
- *OEM in* A_0 *that do not exist in* A_x: The number of over-exposed methods in A_0 that do not exist in A_x anymore.
- *OEM in* A_0 *that are OEM in* A_x: The number of over-exposed methods in A_0 that remain over-exposed in A_x.

Each graph (in the Appendix) describes the profile of the application regarding the evolution of over-exposed methods. Consider the applications SweetHome3D, Jedit, CheckStyle, Jajuk, Jedit and PMD. These applications have the number of over-exposed methods from A_0 reduced by less than 5%, only. **Although the size of these applications increases over time, the number of over-exposed methods found in an early version of these applications remains over-exposed across the analyzed versions.**

On the other end of the spectrum, the applications TuxGuitar and Jmol show their number of over-exposed methods found in A_0 reduced by 76% and 64%, respectively. These two applications have the number of over-exposed methods found in their initial version largely reduced over time.

An interesting result is that the number of methods that become not over-exposed is relatively small for all applications. The 14 applications we considered in this paper have less than 19% of the over-exposed methods found in an early version turned into a non-over-exposed. This measurement indicates that the intuition *giving an accessibility greater than necessary to a method for future usage* holds only for a small portion of the methods.

Figure 4 gives a global estimation of the evolution of over-exposed methods for the 6 versions of the 13 applications (without CheckStyle and Logisim). The graph summed up the four metrics given above. It shows that, from the 15,276 over-exposed methods found in the initial version of the applications and after six successive versions, 71.97% (= 10,994) methods remain over-exposed, 19.71% (= 3,011) are removed and 8.32% (= 1,271) become not over-exposed.

5 Why so many over-exposed methods?

Fully understanding the causes of having an average of 20% of over-exposed methods is difficult. Furthermore, around 70% of these methods are likely to remain over-exposed over time. Many factors related to education, programming culture and habits may explain the rather high number of over-exposed methods. One reason that explains why method accessibility is improperly used and rarely changed may be found in the support offered by programming environments. An exhaustive

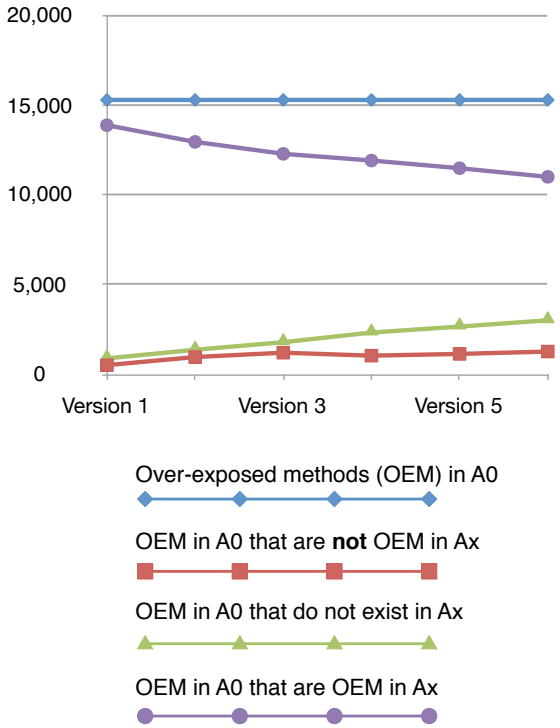

Fig. 4 Cumulative evolution of over-exposed methods

explanation cannot reasonably be provided in this article due to the complexity of the task. Instead of focusing on the root of the problem, we investigate how frequently access modifiers are changed and provide a possible explanation for this.

Code review tools. Code review tools are popular for quality control, and they usually exercise static analysis on a source code. These kinds of tools come with a set of rules that identify anomalies based on the source code.

Popular code review tools for Java are PMD 5.0.1[12], CheckStyle 5.6[13], and FindBugs 2.0[14]. We have reviewed the rules offered by each of these tools, with the purpose of analyzing whether they help identify and reduce the number of over-exposed methods. Surprisingly, none of them offer check rules to identify over-exposed methods. Several rules are about improper usage of method accessibility. For example, the three tools provide a rule to identify protected methods defined in final classes. Having a protected method in a final class is rather meaningless. However, the tools do not assess the accessibility of a method based on the callers of that method.

[12] http://pmd.sourceforge.net
[13] http://checkstyle.sourceforge.net
[14] http://findbugs.sourceforge.net

Refactoring tools. We have reviewed the set of refactorings offered by three popular programming environments for Java, namely Eclipse[15], IntelliJ IDEA[16] and NetBeans Java[17]. These environments offer to practitioners a large set of refactorings to improve the quality of the source code. These three environments support a specific refactoring to change its signature for a given method, its return type, and the order of the parameters. The accessibility may also be changed, and the consistency of the code is verified against the new given accessibility. However, using this refactoring to modify the accessibility is equivalent to directly changing the source code: no suggestion about the optimal method accessibility. During a programming activity, a programming environment makes suggestions about obvious and simple modifications (*e.g.,* unnecessary package imports or variables that are never read). However, over-exposed methods are not reported.

Eclipse automatically generates method stubs[18]. If the generated method is in the same class from where it is called, then the private accessibility is given.

Other potential factors. Other factors besides a poor support of the programming environments may explain the large proportion of over-exposed methods. Although we did not conduct any controlled study, our extensive experience in teaching Java shows that method accessibility in Java is a complex topic. For example, our experience has shown us that engineers and students are often not aware that a private instance method is statically bound or that a `protected` method is in fact visible within its package (and not only to its subclasses). Our feeling is that the Java accessibility model is more complex than, for example, the Ruby accessibility model. One way to verify this is to conduct a similarly study of applications written in Ruby.

Another intuition we have from our teaching experience, is that most students spend effort on the actual behavior of a given method. Students caring about design will typically try to find out how to shorten methods or properly distribute responsibilities. However, method accessibility is apparently rarely considered in the thinking effort. In the future, we plan to monitor programming activities (Ge et al 2012) to verify this intuition.

6 Threats to Validity

Our case-study and its results are subject to validity threats. Since the approach is based on the call graph analysis of the methods, the main threats are related to whether a method is rightfully exposed to a particular interface. Such threats constitute a source of false negatives and false positives.

Effects of other research. Some of the applications we analyzed have also been analyzed in other research experiments. For example, Zoller and Schmolitzky (2012) analyzes SweetHome3D, PMD, FreeMind. Numerous papers analyze JHotDraw,

[15] version Juno (4.2)
[16] version 9.0.4
[17] version 7.0.1
[18] This happens when you call a method that does not exist and you ask Eclipse to automatically generate the missing method.

e.g., Binkley et al (2005). Researchers have a tendency to contact authors to share and validate their findings (as we did with SweetHome3D, Section 8). This means that the design of these applications may have been influenced by previous experiments.

Sampling. In total, we analyzed 30 applications, which represent over 3.6 million lines of code. Identifying these 30 applications is non-trivial. We spent a fair amount of time finding Java applications that (i) have several versions for us to monitor the evolution and (ii) may be imported into the Eclipse programming environment to be processed by our plugin. Having an application that is "compilable" is important since we have to make sure that no errors are present in the application before running our analysis tool. We first naturally opted for the Qualitas Corpus Tempero et al (2010), a popular corpus of 112 software systems. However, many of the applications in that corpus are not in a compilable state. Configuration files are crucial and are not always distributed with the applications. This means that we would have to manually repeat the tedious loop of (i) downloading an application, (ii) manually identifying what the downloaded archive is made of, (iii) importing the application into Eclipse, and (iv) fixing dependencies by downloading missing libraries.

Design. Some methods may be intentionally defined as over-exposed by programmers. Our personal experience and the discussion with some authors (Section 8) show that a number of methods are often left over-exposed to address some possible future need.

Studying software evolution (Section 4) clearly indicates that over-exposed methods found in an early version of an application remain over-exposed. However, programmers believe that some of these over-exposed methods deserve to be visible to an audience larger than necessary. This fact clearly reflects an intuition shared by programmers. Unfortunately, we were not able to measure or even confirm this intuition. Measuring the number of over-exposed methods that are intentionally over-exposed requires (i) carrying out the case-study we conducted for SweetHome3D with the 14 remaining applications and (ii) identifying the authors of each software component and getting in touch with them. The software we have chosen for our case-study is the result of a large community effort, for which traceability of classes and methods may not be carried out in a satisfactory manner (for example, most source code versioning systems for Java operate at file level granularity, therefore extracting information about methods is challenging). Since open source communities are places with a significant turnover of developers, identifying and contacting the primary author of over-exposed method is difficult.

Static analysis. Our approach employs static analysis to identify over-exposed methods. We are thus facing limitations that static analysis imposes on us. There could be callers that are not identified because of an under approximation of the virtual method call resolution (at runtime) or because of the use of callbacks or reflection. This situation could lead to false positives due to missing calls or simply dead code. However, we think that the number of caller methods missed is generally low. This is supported by the fact that Moose, the tool that we used to analyze the source code, implements a call analysis algorithm similar to rapid type analysis (RTA) (Bacon and Sweeney 1996) to construct the call graph of the applications.

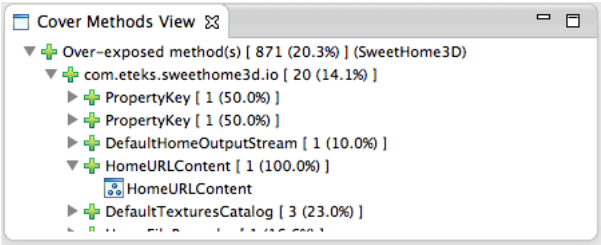

Fig. 5 Methods list

Also, Moose analyzes the presence of keywords such as `this` and `super` to refine the virtual calls. It has been reported that the precision of RTA for resolving virtual method calls is in the range of 70-100% (Liang et al 2001).

7 Support for Practitioners

In order to help software engineers in managing the accessibility of methods, we extended the Eclipse programming environment with *Cover*[19], a plugin to identify and address improper method accessibility.

7.1 Plugin description

Cover takes as input the source code of a Java application. After analyzing it, Cover provides the following output:

- *the list of over-exposed methods* structured along packages and classes of the application;
- *possible refactorings* by indicating the most appropriate accessibility for the current state of the application.

The number of over-exposed methods is given for each reported package and class. For example, Figure 5 reports some results for the SweetHome3D application. Cover indicates that 871 methods are over-exposed, which account for 20.3% of the total number of methods defined in the application. By delving down into the structure of the application, relevant information is given for each of the packages and classes. For example, package `com.eteks.sweethome3d.io` contains 20 over-exposed methods. These 20 methods represent 14.1% of the methods defined in the package. Additionally, over-exposed methods are marked in the source code along with the proper accessibility (Figure 6).

The plugin uses the Java development tools of Eclipse (JDT) to iterate over all the methods and retrieve the calling methods for each method of the application. Retrieving calling methods can be a time-consumming operation. For instance, our plugin takes approximately 11 minutes to analyze SweetHome3D[20]. However, we believe the performance of the method analysis can be improved by adding appropriate caches.

[19] http://sites.google.com/site/santiagoavidal/projects/cover-methods

[20] Experiments were conducted on a MacBook Air, CPU 1.8 GHz Intel Core 5. 4Gb of memory.

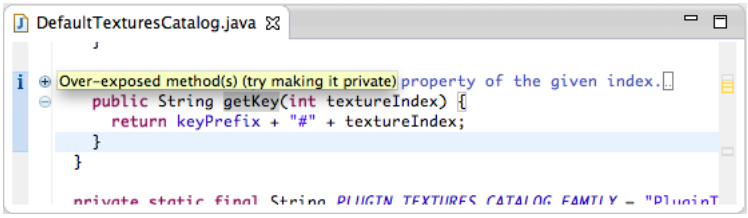

Fig. 6 Refactoring suggestion

Statement	Strongly disagree	Disagree	Neither	Agree	Strongly agree
There is a need for controlling method accessibility			2	4	4
The plugin is easy to use			2	3	5
The plugin is useful				4	6
The suggestions made by the plugin are clear			1	4	5
You will be able to find the over-exposed methods and perform the refactoring without using the plugin	1	5	1	3	

Table 3 Plugin survey.

7.2 Plugin evaluation

With the goal of evaluating if Cover helps developers achieve the task of choosing the most appropriate accessibility for methods, we conducted an experiment with ten PhD students. The experiment was run off-line. All students had previous experience with Java and OOP in an industrial setting. Also, they had access to a tutorial that described how to install and use our plugin.

We assigned to each student the task of refactoring the over-exposed methods of the application Clustermines[21]. Specifically, the students had to go through the methods listed as over-exposed by the plugin and apply the suggested refactorings on the accessibilities of the methods.

After running the experiment we checked that the students refactored all the over-exposed methods identified. Also, the students filled out survey about the plugin and the experiment. The results of this survey are presented in Table 3. For each affirmative statement, the table indicates how many participants chose a level of agreement with it.

While this experiment is not completely comprehensive, it shows that the participants found the plugin easy to use, and that they think that there is a need to change method accessibilities. Specifically, we found that 80% of the participants agree on a need to control method accessibilities (agree+strongly agree). Also, the 80% of the participants agree that the plugin is easy to use. All the students agree on the usefulness of the plugin. The ninety percents of the participants also agree on the clearness of the suggestions of the plugin. Finally, while 30% of the students think that they will be able to find over-exposed methods without the plugin, 60%

[21] http://clustermines.sourceforge.net

disagree or strongly disagree on this point. These observations reinforce the need for a tool like our plugin that helps developers control method accessibilities.

8 Reducing method accessibility

Section 2.3 identified a large proportion of over-exposed methods in about 30 Java applications. We claim that a large portion of those methods may be removed by simply reducing their accessibility modifier. We empirically verified this claim by refactoring an application and assessing that its (observable) behavior did not change.

8.1 Manual Refactoring

SweetHome3D[22] is a 84K LOC application. It comes with 26 unit tests, themselves defining 40 test methods. According to Cobertura, a popular test code coverage for Java[23], the test coverage of SweetHome3D is 70.16%. An application is considered well-tested with a test coverage over 70% (Mockus et al 2009).

Methodology. To measure the impact of reducing the accessibility of over-exposed methods, we conducted the following experiment on SweetHome3D:

1. run all the tests and verify that they all pass
2. run the application and try out the tutorial
3. find the over-exposed methods
4. reduce the accessibility of each over-exposed method to its strict necessary accessibility.
5. recompile SweetHome3D
6. run all the tests and verify that they all pass
7. run the tutorial and look for odd behavior

These steps can be easily applied to other applications. Our refactoring of SweetHome3D source code is available online at `http://bit.ly/SweetHomeRefactoring`.

We ran the tests and looked for odd behavior after changing the accessibility of the methods to check that no method shadows[24] existed (i.e. conflicts between methods or classes with the same name). Odd behavior refers to test failures, visualization errors, or application crashes, among others.

Experiment results. All tests remained green and we did not notice any odd or unexpected behavior. The variation of accessibility is given in the table below:

methods	*before*	*after*	*Δ*
private	945	1,022	+ 8.14%
package	270	620	+ 129.62%
protected	221	227	+ 2.71%
public	4,080	3,647	- 10.61%
over-exposed	682	31	- 95.45%

[22] http://www.sweethome3d.com
[23] http://cobertura.sourceforge.net
[24] http://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html#jls-6.4

The *before* and *after* columns give the number of methods before and after the refactoring. The variation is computed as Δ = (*after* - *before*) / *before*. The over-exposed row corresponds to the number of over-exposed methods.

SweetHome3D defines 5,516 methods, for which 12.36% (= 682) are over-exposed. We removed 95.45% of the over-exposed methods by simply changing the accessibility modifier. The over-exposed methods whose visibility could not be reduced were false positives caused by callbacks and reflective method invocations.

While we cannot generalize the results of this refactoring to other applications, future work could replicate this experiment with a larger set of applications.

8.2 Feedback from the authors

We submitted our refactored version of SweetHome3D to its authors. They expressed great interest in our results because, ensuring a high quality of their products is indeed a strong priority.

Our refactoring addressed many public methods defined in private inner classes. Since private inner classes cannot be accessed from outside the encapsulating class, the authors did not feel it relevant to include these refactorings.

As we discussed earlier for other applications, having over-exposed methods may prepare the application to address future requirements. The authors of SweetHome3D have deliberately over-exposed many methods and constructors for that purpose. The authors prefer to leave such methods untouched. This feedback could lead to an improvement of our plugin. For instance, a future feature is to support filtering criteria, so that the user can exclude certain methods from the analysis.

Using our plugin, we found a number of public methods that could be private. The authors recognized them and agreed with their resolution. The next version of SweetHome3D 4.0 will include parts of our refactoring.

9 Related Work

As far as we are aware of, no large-scale empirical study has been conducted on the accessibilities of methods in object-oriented programming languages. However, some works have identified the uses of access modifiers that are not restrictive enough.

Bouillon et al (2008) present a tool that checks for over-exposed methods in Java applications. 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). This approach does not analyze overridden methods (which can also be over-exposed). 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, no historical analysis is performed.

Müller (2010) 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.

Zoller and Schmolitzky (2012) 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 they employ two software metrics: *Inappropriate Generosity with Accessibility of Types (IGAT)* and *Inappropriate Generosity with Accessibility of Methods (IGAM)*. IGAM is equivalent to our concept of 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 agrees with our observations. Interestingly, this work reports that, on average, 35% of methods are over-exposed[25]. This value is higher than the 20% we measured. We think that this difference is because the authors do take not overridden methods into account.

Steimann and Thies (2009) highlight the difficulties of carrying out refactoring in the presence of non-public classes and methods. Steimann and Thies formalize accessibility constraints in order to check the preconditions of a refactoring (*e.g.,* moving a class to another package requires checking whether the visibility 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. For example, consider the following example:

```
class C1 {
  public void bar() { ... }
}
class C2 extends C1{
  public void foo() { this.bar(); }
}
```

If `C1.bar()` is not used by anyone else in the system, then *protected* is identified by our approach as the ideal visibility for `C1.bar()`. Applying the *change accessibility* refactoring to turn `C1.bar()` into a private method may have a ripple effect of refactoring (*e.g.,* moving `C2.foo()` into `C1`). This change of `C2` may in turn be governed by other constraints, which must be generated as well. A general approach for naming and accessibility for refactoring was later build upon this work (Schafer et al 2012). An Eclipse plugin has also been proposed as an implementation of the constraint-based model of accessibility. Their plugin differs from our since we focus on over-exposure.

Fowler (2002) emphasizes the distinction between public methods and published interfaces. While the changes of a public method of an application can be measured, Fowler's work alerts that changes on interfaces may severely affect other systems that use the application. For this reason, he suggests that the number of published interfaces should be as limited as possible.

Patenaude et al (1999) present the extension of a proprietary source code analysis tool with Java metrics. This extension contains simple metrics related to coupling such as: number of public, private and protected methods, and numbers of calls to a method. After applying the metrics to a group of seven library applications,

[25] We obtained the value 35% by computing the average of the IGAM metric from Figure 3 in Zoller and Schmolitzky (2012).

a very low number of private methods and a majority of public methods were found. However, the reason for those findings are not analyzed.

Briand et al (1999) empirically analyze the relationship between coupling and the rippled effect in object-oriented applications. They determined that classes with high coupling values are more prone to be changed when changes in the public interfaces of classes are performed.

Singh and Kahlon (2011) use static analysis to predict bad smells in code using software metrics. They present two metrics to measure information hiding that are focused on the number of public and private methods of a class respectively. This work found that both metrics are useful in diagnosing smelly classes.

Chowdhury and Zulkernine (2010) analyze the existence of a relationship between different metrics of complexity, coupling and cohesion and security failures. After analyzing several versions of an application they conclude that an important correlation exists between these metrics and vulnerabilities. A similar conclusion is achieved by Singh et al (2012) who also analyze the relation of coupling with software defects.

10 Conclusion and Future Work

Determining the right accessibility when defining a method is key to preserving the right amount of encapsulation and information hiding in object-oriented systems, favoring maintenance and modifiability. We have empirically measured, for a given corpus, that over 20% of the methods are over-exposed. We also found that more of the 70% of the methods of the applications are defined as public (Q1). Also, we have found that libraries have on average more over-exposed methods than plain applications (Q2). Additionally, we found that less than 10% of the over-exposed methods defined in early versions of the applications become non-over-exposed in future versions (Q3).

We have proposed three patterns to identify over-exposed methods in code. We have developed an Eclipse plugin that augments the programming environment with the ability to detect and refactor over-exposed methods.

As future work we plan to:

- refine our analysis by carefully considering the use of reflection and callbacks, increasing the range of analyzable methods;
- perform a study of over-exposed methods in framework-based applications, so as to determine whether our preliminary findings for JFreeChart can be generalized.
- refine our set of accessibility patterns with the aim of identifying the type of information being exposed by methods.
- monitor programming activity to see how often method accessibility is reconsidered by programmers;
- analyze the impact of over-exposed methods on the interfaces of high-level design software elements (*e.g.,* packages, modules, layers, architectural patterns).
- extend the Cover plugin to support the automated refactoring of the over-exposed methods, ensuring the preservation of behavior.

Overall, this work empirically studied the accessibility modifiers of the methods from multiples angles. We analyzed the over-exposing phenomenon in plain

applications and libraries/frameworks. We also measured the impact of software evolution on method accessibility. We argue that these multiple angles provide a clear analysis of the phenomenon.

Acknowledgements We thank Emmanuel Puybaret, principal author of SweetHome3D, for his feedback on our refactoring. We gratefully thank Romain Robbes and Renato Cerro for their feedback on an earlier draft of the manuscript. We also thank Hugo Manterola and Ignacio Orlando for contributing to the development of the plugin. We thank the anonymous reviewers for their comments and suggestions to improve the quality of this work.

This work was partially supported by PIP Project 112-201101-00078 (CONICET) - Argentina and FONDECYT project 1120094 - Chile.

References

Bacon DF, Sweeney PF (1996) Fast static analysis of c++ virtual function calls. In: Anderson L, Coplien J (eds) OOPSLA, ACM, pp 324–341

Binkley D, Ceccato M, Harman M, Ricca F, Tonella P (2005) Automated refactoring of object oriented code into aspects. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pp 27–36, DOI 10.1109/ICSM.2005.27

Booch G (2004) Object-Oriented Analysis and Design with Applications (3rd Edition). Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA

Bouillon P, Grokinsky E, Steimann F (2008) Controlling accessibility in agile projects with the access modifier modifier. In: Paige RF, Meyer B (eds) TOOLS (46), Springer, Lecture Notes in Business Information Processing, vol 11, pp 41–59

Briand LC, Wst J, Lounis H (1999) Using coupling measurement for impact analysis in object-oriented systems. In: ICSM, pp 475–482

Chowdhury I, Zulkernine M (2010) Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, New York, NY, USA, SAC '10, pp 1963–1969, DOI 10.1145/1774088.1774504, URL http://doi.acm.org/10.1145/1774088.1774504

Fowler M (2002) Public versus published interfaces. IEEE Software 19(2):18–19

Ge Xi, DuBose QL, Murphy-Hill E (2012) Reconciling manual and automatic refactoring. In: Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, ICSE 2012, pp 211–221, URL http://dl.acm.org/citation.cfm?id=2337223.2337249

Liang D, Pennings M, Harrold MJ (2001) Extending and evaluating flow-insenstitive and context-insensitive points-to analyses for Java. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, ACM, New York, NY, USA, PASTE '01, pp 73–79, DOI 10.1145/379605.379676, URL http://doi.acm.org/10.1145/379605.379676

Martin RC (2002) Agile Software Development. Principles, Patterns, and Practices. Prentice-Hall

Meyer B (2009) Touch of Class: Learning to Program Well with Objects and Contracts, 1st edn. Springer Publishing Company, Incorporated

Mockus A, Nagappan N, Dinh-Trong TT (2009) Test coverage and post-verification defects: A multiple case study. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, Washington, DC, USA, ESEM '09, pp 291–301, DOI 10.1109/ESEM.2009.5315981

Müller A (2010) Bytecode analysis for checking Java access modifiers. In: Work in Progress and Poster Session, 8th Int. Conf. on Principles and Practice of Programming in Java (PPPJ 2010), Vienna, Austria

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. CACM 15(12):1053–1058, DOI 10.1145/361598.361623, URL http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf

Patenaude JF, Merlo E, Dagenais M, Lagu B (1999) Extending software quality assessment techniques to Java systems. In: IWPC, IEEE Computer Society, pp 49–

Riel A (1996) Object-Oriented Design Heuristics. Addison Wesley, Boston MA

Robbes R, Lungu M, Roethlisberger D (2012) How do developers react to API deprecation? The case of a Smalltalk ecosystem. In: Proceedings of the 20th International Symposium on the Foundations of Software Engineering (FSE'12), pp 56:1 – 56:11, DOI 10.1145/2393596.2393662, URL http://scg.unibe.ch/archive/papers/Rob12aAPIDeprecations.pdf

Schafer M, Thies A, Steimann F, Tip F (2012) A comprehensive approach to naming and accessibility in refactoring Java programs. IEEE Transactions on Software Engineering 38(6):1233–1257, DOI 10.1109/TSE.2012.13

Siegel S, Castellan NJ (1988) Nonparametric Statistics for the Behavioral Sciences, 2nd edn. McGraw-Hill, New York

Singh S, Kahlon KS (2011) Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. ACM SIGSOFT Software Engineering Notes 36(5):1–10

Singh V, Bhattacherjee V, Bhattacharjee S (2012) An analysis of dependency of coupling on software defects. ACM SIGSOFT Software Engineering Notes 37(1):1–6

Steimann F, Thies A (2009) From public to private to absent: Refactoring Java programs under constrained accessibility. In: Drossopoulou S (ed) ECOOP, Springer, Lecture Notes in Computer Science, vol 5653, pp 419–443

Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: A curated collection of Java code for empirical studies. In: Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pp 336 –345, DOI 10.1109/APSEC.2010.46

Thies A, Bodden E (2012) Refaflex: safer refactorings for reflective Java programs. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, ISSTA 2012, pp 1–11, DOI 10.1145/04000800.2336754, URL http://doi.acm.org/10.1145/04000800.2336754

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A (2000) Experimentation in software engineering: an introduction. Kluwer Academic Publishers, Norwell, MA, USA

Zoller C, Schmolitzky A (2012) Measuring inappropriate generosity with access modifiers in Java systems. In: Proceedings of the 2012 Joint Conference of the 22Nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement, IEEE Computer Society, Washington, DC, USA, IWSM-MENSURA '12, pp 43–52, DOI 10.1109/IWSM-MENSURA.2012.15, URL http://dx.doi.org/10.1109/IWSM-MENSURA.2012.15

Table 4 Library / Frameworks analyzed

#	Library/ Framework	Version	#Meth	%Pub	%Prot	%Pac	%Priv	#LOC	#Clss	#Pac
1	Ant	1.8.3	13,517	79.09	7.63	3.68	9.60	178,759	1,730	79
2	Commons-Compress	1.4.1	1,625	73.97	3.32	4.74	17.97	27,730	206	16
3	Commons-Primitives	1.0	4,136	79.64	13.20	4.69	2.47	26,847	505	4
4	Dom4J	2.0	2,938	84.72	10.11	3.00	2.18	25,524	190	14
5	Hibernate	4.1.3	45,348	82.02	5.63	7.55	4.80	390,254	6,741	765
6	JavAssist	3.12	3,362	65.02	5.74	13.27	15.97	42,460	357	17
7	Jericho-HTML	3.2	1,272	67.85	3.69	12.03	16.43	15,032	148	2
8	JFreeChart	1.0.14	8,635	85.63	4.02	5.40	4.96	15,9217	619	37
9	JHotDraw	7.0.6	3,784	80.79	12.29	1.66	5.26	41,577	470	24
10	JUnit	4.10	2,824	79.28	7.26	5.67	7.79	16,185	866	58
11	Log4J	1.2.16	2,132	75.38	12.57	5.53	6.52	27,890	315	22
12	Maven	3.0.4	6,527	66.05	3.26	23.55	7.14	72,063	940	112
13	Struts	2.3.3	15,541	77.12	6.85	10.86	5.17	159,129	2,490	187
14	Tomcat	7.0.27	16,503	82.96	8.54	2.53	5.97	234,945	1,775	112
15	Xalan	2.7.1	10,424	78.99	5.05	10.00	5.97	259,556	1,237	42
	Average		9,238	77.23	7.28	7.61	7.88	111,811	1,239	99.4

#Meth: number of methods; %Pub: number of public methods; %Prot: number of protected methods; %Pac: number of package visible methods; %Priv: number of private methods; #LOC: total number of lines of code; #Clss: number of classes, including inner classes; #Pac: number of packages;

URLs: *Ant:* http://ant.apache.org/; *Commons-Compress:* http://commons.apache.org/compress/; *Commons-Primitives:* http://commons.apache.org/primitives/; *Dom4J:* http://dom4j.sourceforge.net/; *Hibernate:* http://www.hibernate.org/; *JavAssist:* http://www.jboss.org/javassist/; *Jericho-HTML:* http://jerichohtml.sourceforge.net/; *JFreeChart:* http://www.jfree.org/jfreechart/; *JHotDraw:* http://www.jhotdraw.org/; *JUnit:* http://www.junit.org/; *Log4J:* http://logging.apache.org/log4j/; *Maven:* http://maven.apache.org/; *Struts:* http://struts.apache.org/; *Tomcat:* http://tomcat.apache.org/*Xalan:* http://xml.apache.org/xalan-j/

Table 5 Library / Frameworks analyzed

#	Library/ Framework	#OEM	%OEM	%OEM$_a$
1	Ant	3101	22.94	65.64
2	Commons-Compress	389	23.94	61.75
3	Commons-Primitives	1232	29.79	86.7
4	Dom4J	969	32.98	83.47
5	Hibernate	9121	20.11	55.07
6	JavAssist	943	28.05	53.22
7	Jericho-HTML	369	29.01	65.54
8	JFreeChart	2471	28.62	71.5
9	JHotDraw	1150	30.39	64.5
10	JUnit	315	11.15	43.75
11	Log4J	563	26.41	64.05
12	Maven	1017	15.58	54.18
13	Struts	2865	18.44	57.18
14	Tomcat	4859	29.44	61.15
15	Xalan	2633	25.26	58.15
	Average	2133	24.81	63.06

#OEM: number of over-exposed methods; %OEM_a: Over-exposed methods taking into account only analyzable methods.

Table 6 Applications analyzed

#	Application	Version	#Meth	%Pub	%Prot	%Pac	%Priv	#LOC	#Clss	#Pac
1	Argo	0.34	18,248	70.39	9.26	10.91	9.45	235,539	2,568	127
2	Azureus	4.7.12	42,619	75.99	11.42	8.02	4.57	689,989	7,874	476
3	Checkstyle	5.5	3,769	69.04	6.37	10.56	14.04	47,342	1,062	39
4	Cobertura	1.9.4.1	3,313	35.92	1.66	2.96	59.46	50,719	115	19
5	FindBugs	2.0.1	11,326	68.89	2.97	17.91	10.22	129,041	1,765	64
6	FreeMind	0.9	5,980	77.21	5.37	7.26	10.17	56,766	960	48
7	Jajuk	1.9.6	5,781	85.63	2.13	4.07	8.18	95,660	1,065	40
8	JEdit	5.0	7,696	70.80	3.73	10.58	14.89	134,399	1,271	40
9	Jmol	12.2.33	10,355	47.50	7.86	25.63	19.01	176,544	800	66
10	Jstock	1.0.6	3,797	68.00	1.90	4.98	25.13	56,779	799	19
11	Logisim	0.0.1-a	6,601	72.88	3.77	11.89	11.45	57,134	963	43
12	PMD	4.2.6	4,949	71.91	3.15	2.14	22.79	49,648	706	47
13	Portecle	1.7	684	54.68	8.33	3.51	33.48	20,234	188	11
14	Sweet Home 3D	3.5	5,516	73.97	4.01	4.89	17.13	84,632	1,351	9
15	TuxGuitar	1.2	5,111	63.86	5.05	21.64	9.45	48,511	953	78
	Average		9,050	67.11	5.13	9.80	17.96	128,862	1,496	75.07

URLs: *Argo:* http://argouml.tigris.org/; *Azureus/Vuse:* http://azureus.sourceforge.net/; *Checkstyle:* http://checkstyle.sourceforge.net/; *Cobertura:* http://cobertura.sourceforge.net/; *FindBugs:* http://findbugs.sourceforge.net/; *FreeMind:* http://freemind.sourceforge.net/; *Jajuk:* http://jajuk.info/; *JEdit:* http://www.jedit.org/; *Jmol:* http://jmol.sourceforge.net/; *Jstock:* http://jstock.sourceforge.net/; *Logisim:* http://ozark.hendrix.edu/ burch/logisim/; *PMD:* http://pmd.sourceforge.net/; *Portecle:* http://portecle.sourceforge.net/; *SweetHome3D:* http://www.sweethome3d.com/; *TuxGuitar:* http://www.tuxguitar.com.ar/

Table 7 Applications analyzed

#	Application	#OEM	%OEM	%OEM$_a$
1	Argo	3710	20.33	56.48
2	Azureus	6712	15.75	35.25
3	Checkstyle	428	11.36	59.44
4	Cobertura	758	22.88	82.84
5	FindBugs	2753	24.31	59.68
6	FreeMind	1235	20.65	45.04
7	Jajuk	802	13.87	35.87
8	JEdit	1864	24.22	48.43
9	Jmol	1546	14.93	28.12
10	Jstock	743	19.57	50.34
11	Logisim	1017	15.41	32.57
12	PMD	1428	28.85	76.73
13	Portecle	143	20.91	60.08
14	SweetHome3D	682	12.36	31.7
15	TuxGuitar	1235	24.16	42.16
	Average	1670	19.30	49.65

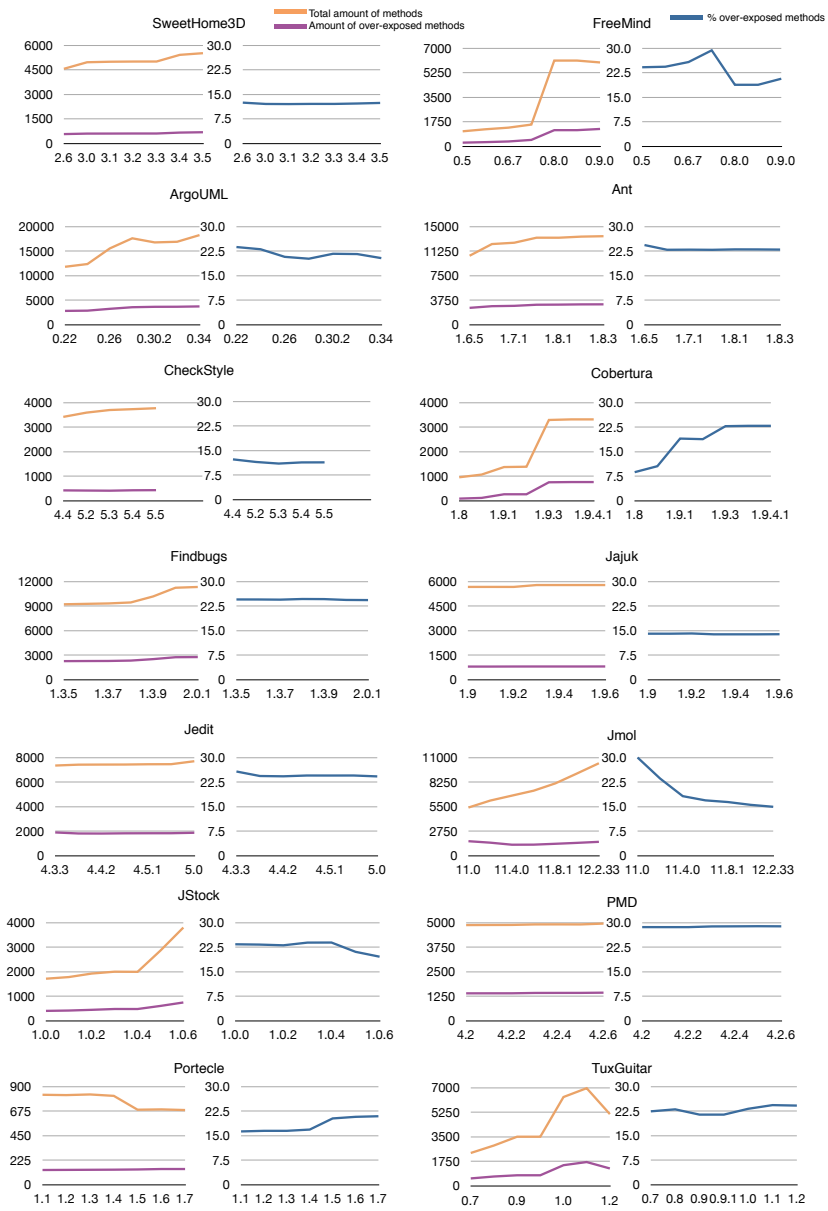

Fig. 7 Evolution of over-exposed methods

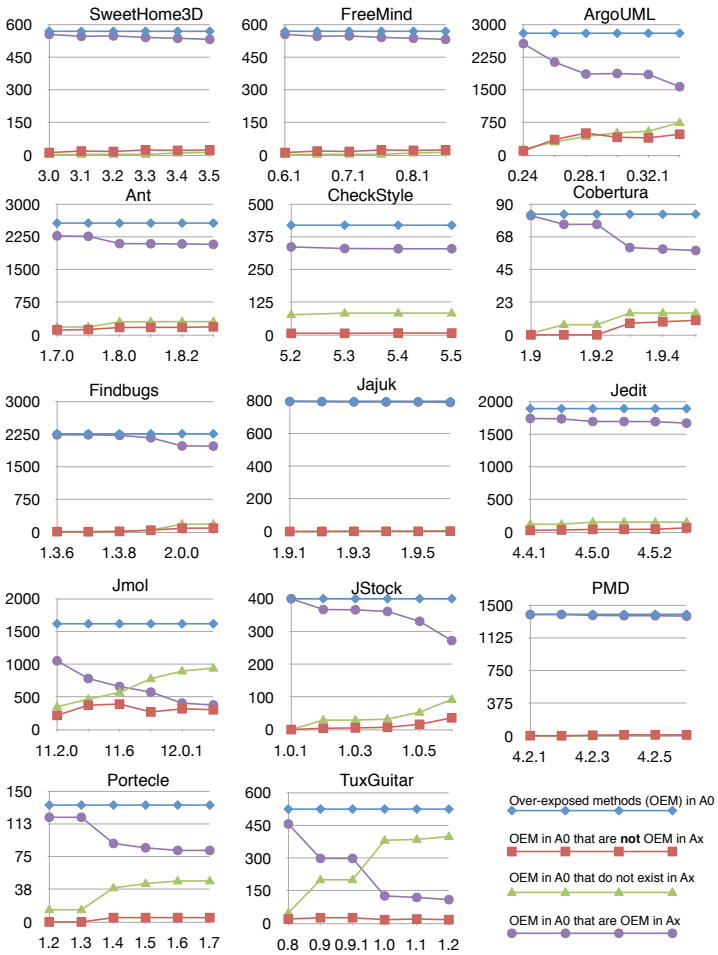

Fig. 8 Evolution of over-exposed methods identified in the first version of each application