Measuring Inappropriate Generosity with Access Modifiers in Java Systems

Christian Zoller, Axel Schmolitzky
Software Engineering and Software Architecture
Department of Informatics, University of Hamburg
Hamburg, Germany
Email: {zoller, schmolit}@informatik.uni-hamburg.de

***Abstract*—Every element of a software architecture, e.g. a subsystem, package, or class, should have a well-defined interface that exposes or hides its subelements according to the principles of information hiding and encapsulation. Similar to other object-oriented programming languages, Java supports defining interfaces on several levels. The accessibility of types, methods, and fields can be restricted by using *access modifiers*. With these modifiers, developers are able to define interfaces of packages and classes tailored for different groups of clients. However, in programming practice, types and members seem to be often declared with too generous access modifiers, i.e. they are accessible by more clients than necessary. This can lead to unwanted dependencies and software quality degradation.**

We developed an approach to measuring the usage of access modifiers for types and methods in Java by defining two new software metrics: *Inappropriate Generosity with Accessibility of Types* (IGAT) and *Inappropriate Generosity with Accessibility of Methods* (IGAM). Furthermore, we created a tool called *AccessAnalysis* that calculates and displays these metrics.

Using AccessAnalysis, we conducted a survey on twelve open source Java projects. The results support our assumption that access modifiers are often chosen more generously than necessary. On average, around one third of all type and method access modifiers fall into this category. Especially top-level types are almost always declared as `public`, so that package interfaces typically expose more types than necessary. Only 2% of all investigated top-level types are encapsulated inside their package.

I. Introduction

Information hiding - hiding design decisions to ease changing them subsequently - is a fundamental concept in software engineering. First described by Parnas in his seminal article [1] and in [2], it serves as a foundation for several guidelines in software design. Encapsulation - hiding implementation details of a software unit from its clients behind an interface [3] [4] - is one specialization of information hiding. Changeability benefits from good encapsulation, since it is easier to replace a unit with a well defined interface and functionality. Comprehensibility benefits from high encapsulation, since it is easier to understand the interaction of software units if they share as little information as possible.

Most object-oriented programming languages support encapsulation via mechanisms of *access control*. With these mechanisms, members of an architectural unit (typically of a class) can either be exported or be hidden. Hidden members are part of the unit's *implementation*, while exported members define the interface to the unit's *clients*. The set of clients might be further divided, e.g. into *external clients* and *subclass clients* [5]. Each set of clients should be kept as small as possible to ensure the lowest possible degree of coupling [6, p. 47: "Few Interfaces Rule"]. The smaller the amount of exported information, the higher the encapsulation [6, p. 48: "Small Interfaces Rule"].

In the *Java programming language* [7] access control is realized with the *access modifiers* `private`, default, `protected` and `public`. With these modifiers, developers are able to define interfaces of packages and classes tailored for different groups of clients to reach an adequate level of encapsulation. However, in programming practice, the potential of the given possibilities seems not always be fully exploited.

In this paper, we define two software metrics for Java to measure the amount of types and methods with an unnecessarily generous access modifier: *Inappropriate Generosity with Accessibility of Types* (IGAT) and *Inappropriate Generosity with Accessibility of Methods* (IGAM). To detect too generous access modifiers and to calculate the metrics, we developed a tool called *AccessAnalysis*[1] [8], a plug-in for the *integrated development environment* (IDE) Eclipse[2]. With AccessAnalysis, we analyzed twelve open source projects to test our hypotheses, that access modifiers are often chosen more generously than necessary.

The remainder of the paper is organized as follows: The following Section II motivates our work and presents the basic concepts. Formal definitions of the metrics IGAT and IGAM, a discussion of their characteristics and a description of the function and usage of AccessAnalysis are given in Section III. The experiment design and its results can be found in Section IV. Section V contains a discussion of the experiment's results together with some considerations regarding the chances and limitations of our approach. In Section VI we take a look at similar works in this area. Section VII concludes the paper.

[1] http://accessanalysis.sourceforge.net
[2] http://eclipse.org

II. Usage of Access Modifiers in Java

In Java, a software system consists of *packages*, which in turn consist of *types*, that are defined by classes and `interfaces`[3]. Classes and `interfaces` contain *members*: fields, methods and nested types.

Types that are not nested inside other types are called *top-level types*, otherwise they are called *member types*. Any top-level type in Java belongs to one specific package. The interface of a package is defined by the top-level types it exports. Top-level types that are marked with the *access modifier* `public` belong to the interface of their package. Top-level types without this modifier are only accessible inside their package; this level of accessibility is called *package-private* or default. Following the principle of maximum encapsulation, every package should only export those types that are needed by clients in other packages. Packages that have such a well-defined interface can act as units of abstraction inside the architecture of a Java system.

Members of `interfaces` are accessible by all clients of the `interface` itself, since all `interface` members are implicitly `public`. The accessibility of class members is controlled by the access modifiers `private`, `protected`, and `public`. Class members declared `private` are only accessible inside their own top-level class. `protected` provides access inside the class' own package and additionally to subclasses from outside the package. Members declared `public` can be accessed by all clients of the class. If the access modifier is omitted, the declared member can only be used inside its own package. Like for top-level types, this access level is called default. For simplification, we use the term default like an access modifier, besides `private`, `protected`, and `public`.

As a result of the possibility to restrict the accessibility of members on the aforementioned levels, classes don't have just one interface, but several for different groups of clients: one for clients inside their own package, one for subclasses outside their own package and one for all other clients. Similar to packages, classes should only export those members that have to be used by their clients. Furthermore, the set of clients that are able to access a member should be kept as small as possible, i.e. the access modifier should be as restrictive as possible [9, p. 67ff.].

A. Software Practice

A look into programming practice shows that the common usage of Java's access modifiers is not as restrictive as it could be. While it is commonly accepted that fields should generally be `private`, the encapsulation of package-internal types or methods is rather uncommon. Especially for top-level classes, the preceding keyword `public` seems to be the quasi-standard. This subjective observation is supported by a preliminary survey we conducted in two professional software projects. In one project, only six of 2,613 types were declared default, in the other one we found two default types out of 355 [10] [11, p. 60ff].

We assume several reasons for this phenomenon. One is programming education: In many relevant textbooks (e.g. [12], [13], [14]) the concept of interfaces is covered in detail on class level by differentiating between `public` and `private`. Yet in most cases it is not transferred to the package level, and default accessibility is only mentioned casually.

Also, lacking tool support might play a role. When Java source code contains a reference to a type or method that needs a higher access level then specified, the compiler raise an error. But except for some academic approaches [15] [16] there are no established tools that raise warnings for types or methods with an unnecessarily generous accessibility. Popular tools such as PMD[4], FindBugs[5], or Checkstyle[6] have some access modifier-related rules, e.g. that fields should be private, but nothing that compares the accessibility of types or methods with their actual usage. PMD even has a rule that advises against the use of default, though it is not clear why this advice is given. Furthermore, in most IDEs, such as Eclipse or NetBeans[7], the standard access modifier for newly created classes or interfaces is not default but `public`.

Another reason might be that developers avoid too restrictive access levels, because they don't want to limit their own possibilities. However, this can lead to unwanted dependencies and lowered software quality.

B. Hypotheses

Our goal is to find an objective, automated, and repeatable way to assess the usage of access modifiers in Java software projects. On this basis, we want to validate the following hypotheses on a selected set of real life software projects:

Hypothesis 1: Access modifiers in Java are often chosen more generously than necessary.

Hypothesis 2: Top-level classes and `interfaces` in Java are almost always declared as `public`, even if they actually don't need to be part of their package's interface.

More generous means that an access modifier gives access to a potentially larger group of clients than another one. For example, `public` is more generous than default. The other way around the access modifier is denoted as *more restrictive*. We will not consider the accessibility of fields, because the rule that these should always be `private` is

[3]To distinguish between the abstract concept "interface" and the Java language construct "`interface`", the language construct is set in typewriter font.

[4]http://pmd.sourceforge.net
[5]http://findbugs.sourceforge.net
[6]http://checkstyle.sourceforge.net
[7]http://netbeans.org

easy enough and can already be checked by existing tools (e.g. PMD, FindBugs).

C. Minimal Access Modifier

To determine whether an access modifier is *more generous than necessary*, we first have to define the meaning of *necessary*. The strict hierarchical structure of access levels in Java leads to the order `private` → default → `protected` → `public`, in which the access modifiers become more generous and, vice versa, more restrictive. The necessary access modifier of a type or method arises from considerations regarding the group of clients that needs access to it. This is a question of interface design that depends on the system's architecture and needs to be answered by the developer for each individual case. Because it also includes considerations about the system's future evolution, there is no definite automatable method for finding the right access modifier. Thus, we use an approximation by defining "necessary" as the current actual usage of the type or method inside the entire software system in its current state. Based on the Java language specification, a most restrictive access modifier can be assigned for each type or method, allowing every use inside the system.

We call this access modifier the *minimal access modifier* of the type or method. The access modifier that is actually assigned to an element in the source code is called *actual access modifier*. In that sense, an element's actual access modifier is *too generous* if it is more generous than its minimal one.

Example: If class A is currently used only by clients from inside its own package, its minimal access modifier is default. If its actual access modifier is `public`, it has a "too generous" access modifier.

III. Two New Metrics for Java

Software metrics are an established approach to assessing attributes of software artifacts in an automated, objective, and repeatable way [17] [18] [19] [20]. To measure the appropriate or inappropriate use of access modifiers, we define two metrics, IGAT and IGAM. The metrics indicate the proportion of those types and methods that have been assigned an unnecessarily generous access modifier.

A. Definition of IGAT

The metric "Inappropriate Generosity with Accessibility of Types" is defined as follows:

$$IGAT(U,P) = \begin{cases} 0, & \text{if } |T(U)| = 0 \\ \dfrac{|T^*(U,P)|}{|T(U)|}, & \text{else} \end{cases} \quad (1)$$

where

- P is the source code of all compilation units of a Java program;
- U with $U \subseteq P$ is a source code subset of P (e.g. the entire source code, a source folder, a package or a type declaration);
- $T(U)$ is the set of all declared types in U and $|T(U)|$ their number;
- $T^*(U,P)$ with $T^*(U,P) \subseteq T(U)$ is the set of all declared types in U with a too generous access modifier and $|T^*(U,P)|$ their number. Here, *too generous* is defined compared to the minimal access modifiers, which have to be determined based on the actual usage of the types inside P.

Thus, our calculation starts with the determination of minimal access modifiers based on the entire source code P. Afterwards, the proportion of types with too generous access modifiers is calculated for U, with U being the source code subset to be measured.

Example: Figure 1 schematically shows the Java project P three times. P is divided into three packages with a total of nine type declarations. The arrows represent dependencies between types, in the sense that the starting point of an arrow marks the using type and the arrowhead the used type. Let the gray-colored types be the ones that are identified to have a too generous access modifier. Depending on the subset U_i of P for which the IGAT value has to be calculated, a certain number of types $|T(U_i)|$ and a certain number of types with a too generous access modifier $|T^*(U_i,P)|$ have to be accounted. The set P stays the same in all three cases.

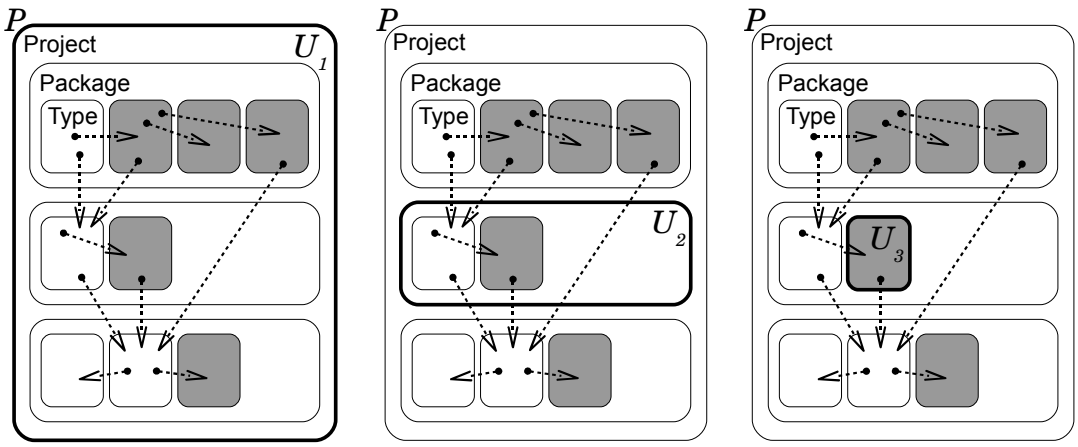
B. Definition of IGAM

Analog to the type metric IGAT, for methods the metric "Inappropriate Generosity with Accessibility of Methods" is defined as follows:

$$IGAM(V,P) = \begin{cases} 0, & \text{if } |M(V)| = 0 \\ \dfrac{|M^*(V,P)|}{|M(V)|}, & \text{else} \end{cases} \quad (2)$$

where

- P is again the source code of all compilation units of a Java program;
- V with $V \subseteq P$ is a source code subset of P (e.g. the entire source code, a source folder, a package, or a type or method declaration);
- $M(V)$ is the set of all declared methods in V and $|M(V)|$ their number;
- $M^*(V,P)$ with $M^*(V,P) \subseteq M(V)$ is the set of all declared methods in V with a too generous access modifier and $|M^*(V,P)|$ their number. Again, *too generous* is defined compared to the minimal access modifiers, which have to be determined based on the actual usage of the methods inside of P.



$|T(U_1)| = 9$ $|T^*(U_1, P)| = 5$ $IGAT(U_1, P) = \frac{5}{9}$

$|T(U_2)| = 2$ $|T^*(U_2, P)| = 1$ $IGAT(U_2, P) = \frac{1}{2}$

$|T(U_3)| = 1$ $|T^*(U_3, P)| = 1$ $IGAT(U_3, P) = \frac{1}{1}$

Figure 1. IGAT calculation

C. Determination of Minimal Access Modifiers

In contrast to basic software metrics, such as Lines of Code (LOC) or Cyclomatic Complexity [21], the measurement cannot be performed by only locally analyzing the object itself (U or V), but the context (P) has to be taken into account. The minimal access modifier of a type or method is derived from all existing references to the particular element in the entire source code of the system. Every single use requires an access level that is determined by the relation between the client and used element. For example, if a method is used inside its surrounding class, the required access level is `private`; if it is used by another class inside the same package, the required access level is default.

The minimal access modifier of a type equates to the most generous required access level from all references. For methods, further rules have to be taken into account: a) `interface` methods are always `public`, b) main methods are always `public`, c) methods that override, implement, or hide other methods cannot reduce their accessibility. *Reference* means every possible direct or indirect use of the particular type or method that requires access rights. Examples for type references are declaring a local variable of a particular type, extending a type or declaring a bound for a wildcard type. A reference to a method can be, for example, its call or its overriding. Due to Java's static type system, references to a type or method can be found by a static source code analysis. This only works as long as the whole source code of the analyzed system is available and mechanisms of *reflection* are not used. The latter can lead to type or method use that cannot be detected by reading the source code. This problem affects every technology that is based on static source code analysis. Even though there exist some approaches that address these problems [22] [23], we will not consider reflection in the following, beside one exception: test classes and methods called by the JUnit framework[8].

D. The AccessAnalysis Tool

For calculating the presented metrics, we developed AccessAnalysis, a plug-in for the Eclipse IDE. AccessAnalysis determines minimal access modifiers of types and methods by analyzing the source code of one or more projects. Based on its analysis, it calculates IGAT and IGAM for each project, source folder, package, type, and method (for methods only IGAM). The results are shown in a tree-like table (Figure 2). Moreover, actual and minimal access modifiers are compared to each other and deviations are highlighted. In case of a type or method that is not used at all in the analyzed projects, the pseudo access modifier not used is assigned as minimal access modifier. When calculating the metrics, not used is considered to be more restrictive than any other access modifier. AccessAnalysis handles constructors like methods. Member types and top-level types are handled equally, except for the fact that minimal and actual access modifiers of member types can also be `private` or `protected`.

E. Special Cases

The *JUnit test framework* is very common and probably used by almost every Java project. JUnit test classes and methods need to be `public` for being called by the framework, even if there are no static references to these classes or methods anywhere else in the source code. This means

[8] http://junit.org

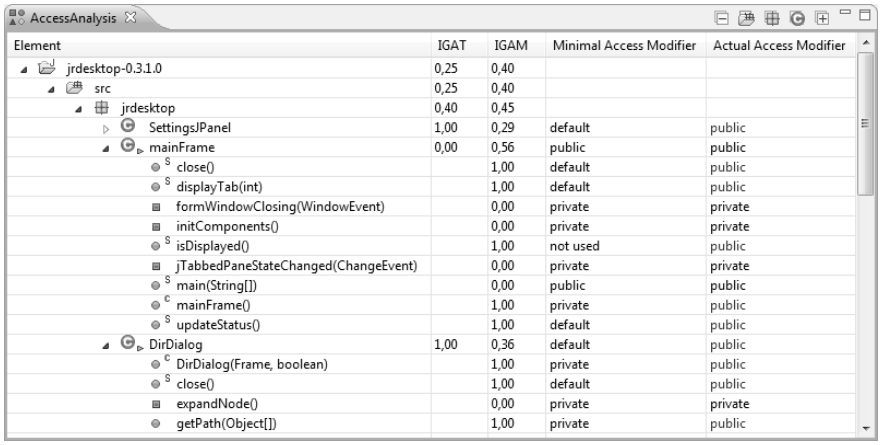


Figure 2. Display of results in the Eclipse Plug-in AccessAnalysis

that their `public` access modifier might be considered as too generous by following the rules of minimal access. Therefore AccessAnalysis contains some exceptional rules for JUnit. By default, every JUnit test class and method gets `public` as its minimal access modifier. This feature works with JUnit 3.x (inheritance- and naming convention-based) and JUnit 4.x (annotation-based), and can be turned off in the plug-in preferences.

Beyond that, custom constraints for exceptional rules can be defined, e.g. for dedicated API types or methods. Therefore custom *annotation types* have to be defined and mapped to an access modifier in the plug-in preferences. The minimal access modifier of a type or method marked with such an annotation will be at least the specified one.

These two features, the JUnit and the annotation constraints, introduce a further issue into the analysis results. Without these features, the minimal access modifier is only determined by the language rules and always the least generous access modifier that is necessary to ensure a compilable source code. This means that the actual access modifier of a type or method can never be more restrictive than the minimal one, as long as the entire source code has no compiler errors. But with JUnit and annotation constraints, it becomes possible, that an actual access modifier is more restrictive than the corresponding minimal one. This can be the case, when the analyzed source code itself doesn't comply with the preset constraints. This fact has to be considered when interpreting the analysis results.

IV. The Survey

To test our hypotheses from Section II-B, we conducted a survey on twelve open source projects (Table 1). Some of them are just well-known, popular open source software, while we found others on the website *sourceforge.net*, a popular hosting platform for open source software projects. The project sizes range from 39 (JDepend) to 1,104 top-level types (FindBugs).

A. Selection and Preparation of Measured Projects

Because AccessAnalysis is based on static source code analysis, only applications with available source code were candidates for our survey. Beyond that, the measured

		Pkgs.	Types	Meth.	LOC
BlueJ 3.0.7	Java Learning IDE – http://bluej.org	56	743	8,483	99,624
Cobertura 1.9.4.1	Java test coverage tool – http://cobertura.sourceforge.net	31	130	3,478	54,334
DoctorJ 5.1.2	Javadoc analyzer tool – http://www.incava.org/projects/java/doctorj	18	246	4,196	33,558
FindBugs 2.0.0	Java error analyzer – http://findbugs.sourceforge.net	74	1,104	10,281	115,056
FreeCol 0.10.3	Strategy game – http://www.freecol.org	51	664	7,883	106,412
FreeMind 0.9.0	Illustration editor – http://freemind.sourceforge.net	48	445	5,704	53,740
JabRef 2.7.2	Bibliography management – http://jabref.sourceforge.net	58	611	4,912	77,909
JDepend 2.9.1	Java metric tool – http://www.clarkware.com/software/JDepend.html	5	39	453	3,547
jrDesktop 0.3.1.0	Remote desktop control – http://jrdesktop.sourceforge.net	11	56	908	11,321
PDFsam 2.2.1	PDF tool – http://www.pdfsam.org	86	299	2,444	26,058
PMD 4.3	Java error analyzer – http://pmd.sourceforge.net	89	761	5,953	62,618
Sweet Home 3D 3.4	Illustration editor – http://www.sweethome3d.com	10	209	4,241	71,453

Table I
Analyzed open source projects and their sizes in packages, types, methods and non-blank/non-comment lines of code (LOC)

projects had to fulfill several more requirements to ensure that the results are valuable:

1) Since AccessAnalysis is up to now only able to analyze plain Java source code, the application need to be written in Java. Even *Java Server Pages* (JSP)[9] or other programming languages running on the *Java Virtual Machine* (e.g. Groovy) are not allowed.
2) The project needs to be a stand-alone application and no library or framework. This is supposed to ensure that the majority of the included types and methods are for using inside the application itself and not for third-party applications.
3) The project should not use reflection, neither in its own code nor in applied frameworks.
4) The source code must not show compiler errors when imported into Eclipse and after all dependencies to third-party libraries were satisfied.

The mentioned requirements are fulfilled by all of the chosen projects except for the following points:

- BlueJ, JabRef, and Sweet Home 3D contain published APIs for extensions. We added annotations to the dedicated API types and methods according to their documentations[10,11,12] and specified corresponding annotation constraints.
- PMD, a popular rule-based source code analyzer for Java, loads its rule classes via reflection. Here, too, we added annotations to these classes and specified an annotation constraint.
- Cobertura, FindBugs, and PMD implement tasks for the build tool Ant[13] that are called via reflection. Again, we used annotation constraints here.

Some of the projects also feature static source analysis and include sample code to test their own analysis. Because such source code parts do not belong to the application itself, we removed them before our analysis. To give an impression of the analysis' performance, for the largest project (FindBugs) it took around half a minute on our experimentation laptop[14].

B. Results

Figure 3 shows the IGAT and IGAM results for the entire source codes of the individual projects, i.e. the results for cases where P and U both present the whole project. We see that a significant number of access modifiers are more generous than necessary. IGAT results range from 14% (Sweet Home 3D) to 51% (JDepend), IGAM from 25% (FreeMind) to 63% (DoctorJ). On average, around one third

[9]http://www.oracle.com/technetwork/java/javaee/jsp/index.html

[10]http://www.bluej.org/doc/extensionsAPI/

[11]http://sourceforge.net/apps/mediawiki/jabref/index.php?title=Getting_started_with_JabRef_plugin_development

[12]http://www.sweethome3d.com/pluginDeveloperGuide.jsp

[13]http://ant.apache.org/manual/develop.html

[14]Dell Latitude E6420, Intel Core i5-2520M CPU @ 2.50GHz, 8 GB RAM, Solid State Disc, Windows 7 (64 Bit), Eclipse Indigo

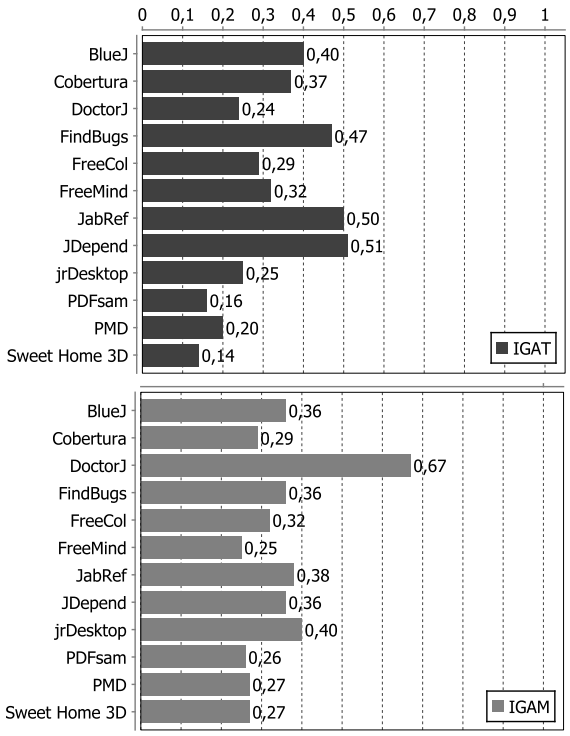


Figure 3. Overall IGAT and IGAM results for analyzed open source projects (U, V, P: entire project)

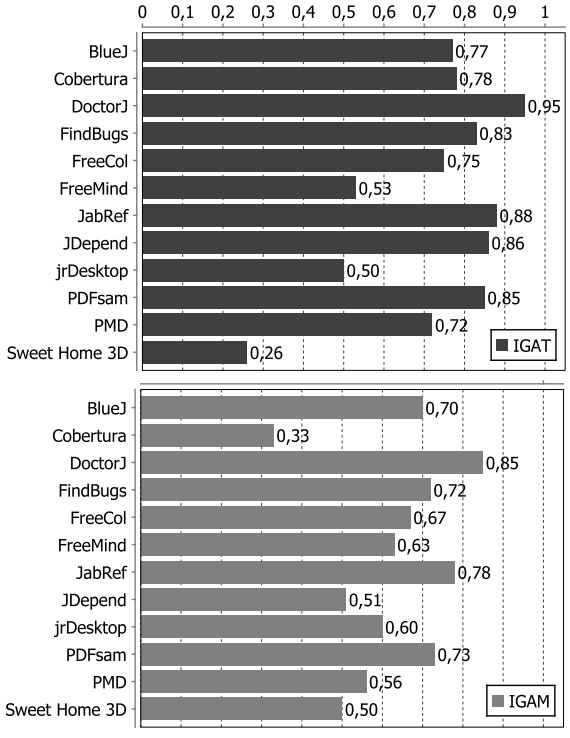
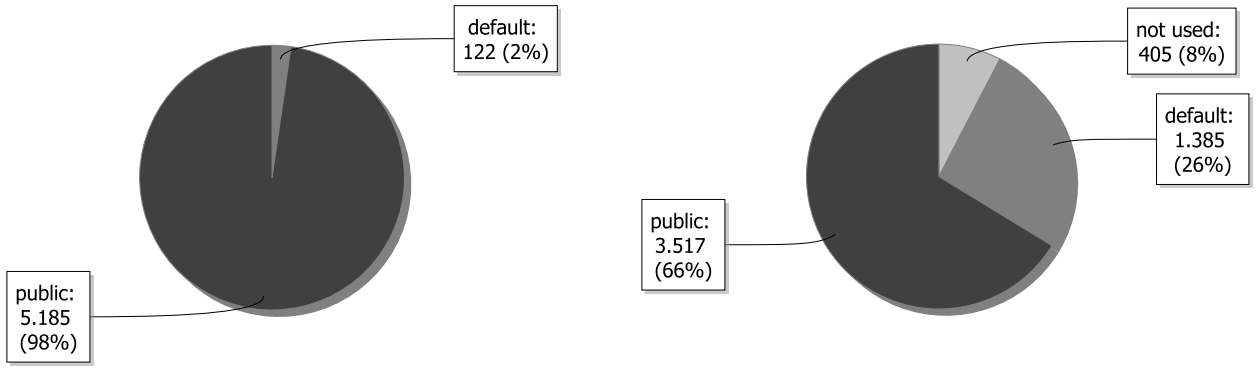


Figure 4. Modified IGAT and IGAM results for analyzed open source projects (P: entire project, U: Types with minimal access modifier not `public`, V: Methods with minimal access modifier not `public`)



(a) Actual Access Modifiers

(b) Minimal Access Modifiers

Figure 5. Total distribution of access modifiers of top-level types in all analyzed projects

of both, types and methods, have access modifiers that are too generous.

It has to be noticed that the metric results are limited to the proportion of types or methods where it is actually possible to specify a too generous access modifier, i.e. where the minimal access modifier is not `public` anyway. This also depends on the package structure of the particular project. If the project consists of many small packages, it is more likely to be necessary to have a `public` type or method than in cases of fewer, but larger packages. This makes it hard to compare the results for completely different projects or to define a general critical threshold. For example, PDFsam has the lowest average ratio of top-level types to packages (3.5) and relative good results compared to projects such as BlueJ, DoctorJ, or FindBugs with more top-level types per package (> 13.0). But these values do not always correlate. For example, Sweet Home 3D has the largest packages on average (20.9 top-level types/package), but the best IGAT and a comparatively good IGAM result, while Cobertura with a top-level type/package-ratio of 4.2 has a much higher IGAT result.

To eliminate the influence of the package structure, it might be useful to limit the results to those types and methods where the minimal access modifier is not `public`. This is accomplished in Figure 4. This presentation strongly supports Hypothesis 1, that in general access modifiers are often chosen more generously than necessary. On average, almost three-fourth of the types and almost two-thirds of the methods with minimal access modifiers more restrictive than `public` have actual access modifiers that are too generous.

The fact that types are even more affected than methods leads us to Hypothesis 2, that especially top-level types are hardly ever hidden in their packages. Figure 5(a) shows that only a small portion of 2% of all top-level types in the analyzed projects has default as actual access modifier. However, there are around ten times more top-level types that actually can be default according to their actual usage in their respective system, see Figure 5(b). This confirms Hypothesis 2. The proportion of actual default top-level types is in most projects approximately the same. Only jrDesktop has a higher proportion of around 11%, and PDFsam has absolutely no default top-level types. Another point that supports our assumption is the fact that Sweet Home 3D has the best IGAT result. That project has an uncommonly high proportion of member types (56% of all types), and their access modifiers seem to have been chosen more carefully.

V. Discussion

Our hypotheses were that in real life Java projects access modifiers are often chosen more generously than necessary and that this affects especially top-level types. The results of our survey confirm these assumptions. Although our selection of analyzed projects could not be seen as representative, our survey shows that the drift between accessibility and actual usage of types and methods does exist in Java projects. Furthermore, the wide disregard of packages as units of encapsulation was found in all analyzed projects.

A. Chances and Limitations

Access modifiers which follow the actual usage requirements make the addressed client group explicit and thus increase the comprehensibility of software. Restrictive interfaces protect from unnecessary dependencies and thus make it easier to change individual parts of a software. The metrics IGAT and IGAM can be used to monitor the evolution of interfaces in a Java software project.

Source code units with high IGAT or IGAM values can be selected for an extra review of their interfaces and designated for refactoring. An automated minimization of accessibilities seems possible, but not advisable, because in individual cases there might exist reasons for using a more generous access modifier than the minimal one. For example, one obvious reason for a too generous access modifier is that a client that needs the declared accessibility still has

to be coded. In projects that are not developed as stand-alone applications, but as libraries or frameworks, many interfaces will surely contain elements that are not used in the project itself. In the same way, when developing a strongly component-based system where the individual components are supposed to be reused in other projects, the interface design will not exclusively follow aspects of minimization.

The minimal access modifier of an element might even exceed the appropriate access level. This could be the case when an insufficient encapsulation is already exploited by clients. However, the minimal access modifier can serve as a useful reference for the developer to decide which level of actual accessibility is required.

The determination of minimal access modifiers based on a fixed source code set (P) makes sense only if the usage of the associated element is limited to this source code set. In some cases it might be necessary to expand P to the source code of associated projects. Also, further documents, such as JSPs or framework configuration files, might have to be included.

The presentation of the results revealed one disadvantage of the metrics IGAT and IGAM: the results of completely different projects cannot be easily compared, since they depend strongly on their package and class structure. One way of addressing this problem is to limit the metric calculation to those types and methods that actually offer means of reducing accessibility, i.e. their minimal access modifier is not `public`; as presented in Fig. 4. But then the metrics lose some of their simplicity, because the calculation is not as easily comprehensible as if the metric represents a proportion of the total number of types/methods.

B. Transfer to Other OO Languages

The transfer of the introduced concepts to other object-oriented programming languages might be desirable, but it is only feasible under certain conditions. First of all, the type system of the language has to allow for a complete static analysis of all dependencies. Moreover, the principle of the minimal access modifier is based on the strict hierarchical order of access modifiers in Java. Such an order cannot be derived in all object-oriented languages. For example, C# has the orthogonal access levels `internal` and `protected`. `internal` allows access within the same assembly (unit of deployment), while `protected` allows access from all subclasses [24, § 10.5]. An adequate agreement would be required to specify whether `internal` or `protected` is the minimal access modifier for elements that are only used by subclasses within their own assembly.

VI. Related Work

A much shorter predecessor of this paper has been published in German [10]. Concepts and characteristics of the metrics IGAT and IGAM are presented in detail in [11]. Usage and inner workings of AccessAnalysis are described in [8].

Bouillon et al. [15] developed an Eclipse plug-in that is able to determine minimal access modifiers for Java as well, but only for methods. This plug-in doesn't calculate any metrics, but offers tool support for the selection of access modifiers during development.

Müller [16] developed a tool that generates reports about several Java modifiers based on byte code analysis. It also detects access modifiers that are too generous, but again only for methods (and fields).

Like us, Bouillon et al. and Müller evaluated their approach based on a couple of open source projects and gained similar results, i.e. that access modifiers are often chosen more generously than necessary.

The encapsulation of methods and fields is subject of two metrics from the MOOD catalog by Brito e Abreu [25]: "Method Hiding Factor" and "Attribute Hiding Factor". These indicate the proportion of non-public methods and fields in all classes. Cao and Zhu [26] expand these metrics by including the number of classes that have access to an individual method or attribute.

Tempero conducted a study [27] of 100 open source java systems, focusing on unused external type members, i.e. fields and methods that are accessible from outside the type but never used there. He found "a surprisingly high number of such members." In contrast to our approach, he did not distinguish between the different access levels and he only examined the usage of type members and not of types themselves. In a second study [28], of again 100 open source java systems, he examined how consequently the rule that fields should always be `private` is followed. The result was that "it is not uncommon [...] to declare non-private fields, but then not take advantage of that access." Although Tempero's studies have different focuses than our, his results point in the same direction. It seems that many Java systems contain elements that are designated for actually inexistent use.

VII. Conclusion

We formulated and motivated the hypotheses that a) access modifiers in Java are often chosen more generously than necessary and b) that this affects especially top-level types (classes and interfaces). To define what *more generous than necessary* means, we introduced the concept of the minimal access modifier. The minimal access modifier of a type or method is the most restrictive access modifier that allows all existing references to the associated type or method in the entire source code of its system. On that basis, we defined the metrics IGAT and IGAM as measures for deviations between minimal and actual access modifiers.

We accomplished the determination of minimal access modifiers and the calculation of metrics in the Eclipse plug-in AccessAnalysis. AccessAnalysis considers all possible

references defined by the Java Language Specification and also respects special rules for JUnit test classes and methods as well as custom exceptional rules specified by annotations. Using AccessAnalysis, we conducted a survey on twelve open source Java projects. The results of the survey support our hypotheses. Around one third of all types and methods in the analyzed projects have a more generous access modifier than necessary according to their actual usage in their projects. Furthermore, only 2% of all top-level types have assigned default (package-private) accessibility, which shows that packages are widely disregarded as units of encapsulation.

Carefully designed interfaces that expose no more elements than necessary are a key ingredient of good software architecture. The presented metrics can help to monitor interface evolution in Java projects. Further tool support for the handling of access modifiers during the development process, as proposed by Bouillon et al. [15], might be useful, but need not only pay attention to class members. Also, the accessibility of top-level types has to be taken into account.

Future work on our approach contains the expansion of AccessAnalysis to the analysis of further source code documents like JSPs or framework configuration files. In addition, a deeper evaluation of the metrics' usefulness in the software development process would be of interest.

References

[1] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[2] ——, "Information distribution aspects of design methodology," in *Proc. of IFIP Congress 71*, vol. 1, Ljubljana, Yugoslavia, Aug. 23–28, 1971, pp. 339–344.

[3] A. Snyder, "Encapsulation and inheritance in object-oriented programming languages," in *Conference proc. on Object-oriented programming systems, languages and applications (OOPSLA '86)*, Portland, OR, USA, Sep. 29–Oct. 2, 1986, pp. 38–45.

[4] B. Liskov, "Data abstraction and hierarchy," in *Addendum to the proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87)*, Orlando, FL, USA, 1987, pp. 17–34.

[5] R. Wirfs-Brock and B. Wilkerson, "Object-oriented design: a responsibility-driven approach," in *Conference proc. on Object-oriented programming systems, languages and applications (OOPSLA '89)*, New Orleans, LA, United States, Oct. 1–6, 1989, pp. 71–75.

[6] B. Meyer, *Object-oriented Software Construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1997.

[7] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Upper Saddle River, USA: Addison-Wesley, 2005.

[8] C. Zoller and A. Schmolitzky, "Accessanalysis — a tool for measuring the appropriateness of access modifiers in java systems," in *Proc. of 12th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2012)*, Riva del Garda, Italy, Sep. 23–24, 2012.

[9] J. Bloch, *Effective Java*, 2nd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2008.

[10] C. Zoller and A. Schmolitzky, "Zwei Metriken zum Messen des Umgangs mit Zugriffsmodifikatoren in Java," in *Proc. of Software Engineering 2011 – Fachtagung des GI-Fachbereichs Softwaretechnik (SE2011)*, Karlsruhe, Germany, Feb. 21–25, 2011.

[11] C. Zoller, "Ein Ansatz zur Messung der Kapselung in Java-Systemen," diploma thesis, Universität Hamburg, Hamburg, Germany, 2010. [Online]. Available: http://swt-www.informatik.uni-hamburg.de/uploads/media/Diplomarbeit_Christian_Zoller.pdf

[12] D. Barnes and M. Kölling, *Objects First with Java – A Practical Introduction using BlueJ*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2008.

[13] C. Horstmann and G. Cornell, *Core Java – Volume I: Fundamentals*, 8th ed. Upper Saddle River, USA: Prentice Hall, 2007.

[14] W. Savitch, *Absolute Java*, 4th ed. Boston, USA: Pearson Education International, 2009.

[15] P. Bouillon, E. Großkinsky, and F. Steimann, "Controlling accessibility in agile projects with the access modifier modifier," in *Proc. of TOOLS (46)*, 2008, pp. 41–59.

[16] A. Müller, "Bytecode analysis for checking java access modifiers," in *Proc. of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)*. Vienna, Austria: CEUR Workshop Proc., Sun SITE Central Europe, Sep. 16, 2010. [Online]. Available: http://ceur-ws.org/Vol-692/paper6.pdf

[17] V. Basili and D. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Softw. Eng.*, vol. 10, no. 6, pp. 728–738, 1984.

[18] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[19] N. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*, 2nd ed. Boston, MA, USA: PWS Publishing Co., 1997.

[20] H. Zuse, *A Framework of Software Measurement*. Hawthorne, NJ, USA: Walter de Gruyter & Co., 1997.

[21] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[22] B. Livshits, J. Whaley, and M. Lam, "Reflection analysis for Java," in *Proc. of the 3rd Asian Symposium on Programming Languages and Systems (APLAS'05)*, Tsukuba, Japan, Nov. 2005.

[23] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *Proc. of the 33rd International Conference on Software Engineering (ICSE '11)*, Honolulu, HI, USA, May 21–28, 2011, pp. 241–250.

[24] *C# Language Specification*, ECMA International Std. Standard ECMA-334, Rev. 4, 2006.

[25] F. Brito e Abreu and R. Carapuça, "Object-oriented software engineering: Measuring and controlling the development process," in *Proc. of the 4th International Conference on Software Quality (QSIC '94)*, McLean, VA, USA, Oct. 3–5, 1994.

[26] Y. Cao and Q. Zhu, "Improved metrics for encapsulation based on information hiding," in *The 9th International Conference for Young Computer Scientists (ICYCS 2008)*, Zhang Jia Jie, Hunan, China, Nov. 18–21, 2008, pp. 742–747.

[27] E. Tempero, "An empirical study of unused design decisions in open source java software," in *Proc. of the 15th Asia-Pacific Software Engineering Conference (APSEC 2008)*, Beijing, China, Dec. 3–5, 2008, pp. 33–40.

[28] ——, "How fields are used in java: An empirical study," in *Proc. of Australian Software Engineering Conference 2009 (ASWEC '09)*, Gold Coast, Queensland, Australia, Apr. 14–17, 2009, pp. 91–100.