AccessAnalysis — A Tool for Measuring the Appropriateness of 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*—Access modifiers allow Java developers to define package and class interfaces tailored for different groups of clients. According to the principles of information hiding and encapsulation, the accessibility of types, methods, and fields should be as restrictive as possible. However, in programming practice, the potential of the given possibilities seems not always be fully exploited.**

***AccessAnalysis*[1] is a plug-in for the Eclipse IDE[2] that measures the usage of access modifiers for types and methods in Java. It calculates two metrics, *Inappropriate Generosity with Accessibility of Types* (IGAT) and *Inappropriate Generosity with Accessibility of Methods* (IGAM), which represent the degree of deviation between actual and necessary access modifiers. As an approximation for the necessary access modifier, we introduce the notion of *minimal access modifiers*. The minimal access modifier is the most restrictive access modifier that allows all existing references to a type or method in the entire source code of a system. AccessAnalysis determines minimal access modifiers by static source code analysis using the build-in Java DOM/AST API of Eclipse.**

I. Introduction

Information hiding and encapsulation are fundamental concepts in software engineering. Most object-oriented programming languages support encapsulation via mechanisms of *access control*. With these mechanisms, members of an architectural unit (typically 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. Each set of clients should be kept as small as possible to ensure the lowest possible degree of coupling. The smaller the amount of exported information, the higher the encapsulation [1, p. 46–53].

A. Package and Type Interfaces in Java

The *Java programming language* [2] supports interface building via access control on two levels: the package and the class level. The interface of a package consists of all top-level types in the package that are marked with the access modifier `public`. Top-level types are classes and `interfaces` (to distinguish between the abstract concept "interface" and the Java language construct "`interface`" , the language construct is set in typewriter font) that are not nested inside other types. Top-level types without `public` modifier are accessible only inside their package; this level of accessibility is called *package-private* or default.

Types, in turn, contain members: fields, methods, and nested types. While `interface` members are implicitly `public`, the accessibility of class members is, again, controlled by access modifiers. Class members declared `private` are accessible only 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 be used only inside its own package. As for top-level types, this access level is called package-private or default. For simplification, in the following 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 export only 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 [3, p. 67–70].

B. 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.

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

One reason might be lacking tool support. 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 [4] [5] there are no established tools that raise warnings for types or methods with an unnecessarily generous accessibility. Popular tools such as PMD[3] or FindBugs[4] 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 *integrated development environments* (IDE), such as Eclipse or NetBeans[5], the standard access modifier for newly created classes or interfaces is not default but `public`.

C. Our Approach

With *AccessAnalysis* we developed a tool to measure the (in)appropriate usage of access modifiers in Java. It compares the accessibility of types and methods with their actual usage inside the entire project source code. Thus, it primarily aims on measuring stand-alone applications and less on frameworks or libraries containing many types and methods intended for use in third-party projects.

The remainder of the paper is organized as follows: The following Section II presents the basic concepts, the metrics calculated by AccessAnalysis and the usage of the tool. Section III gives an overview of the tool's architecture and its inner workings. In Section IV we go further into the Java language rules and exceptions that affect the analysis, and finally conclude in Section V.

II. Measuring the Usage of Access Modifiers in Java

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/method. The access modifier that is actually assigned to an element in the source code is called *actual access modifier*. Following these definitions, 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.

A. Metric Definitions

Software metrics are an established approach for assessing attributes of software artifacts in an automated, objective, and repeatable way [6] [7]. To measure the (in)appropriate usage of access modifiers, we define two metrics, *Inappropriate Generosity with Accessibility of Types* (IGAT) and *Inappropriate Generosity with Accessibility of Methods* (IGAM), that indicate the proportion of those types and methods that have been assigned an unnecessary generous access modifier.

Definition 1 (IGAT):

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

where P is the source code of all compilation units of a Java program, U the source code subset to be measured (e.g. the entire source code, a source folder, a package or a type declaration), $|T(U)|$ the total number of declared types in U and $|T^*(U,P)|$ the number of types in U with a too generous access modifier, determined by the actual usage inside P. In cases with no type declarations in U, $IGAT(U,P)$ is defined as 0.

Example: Fig. 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.

Definition 2 (IGAM):

$$IGAM(V,P) = \frac{|M^*(V,P)|}{|M(V)|} \tag{2}$$

where P is again the source code of all compilation units of a Java program, V the source code subset to be measured (e.g. the entire source code, a source folder, a package or a type or method declaration), $|M(V)|$ the total number of declared methods in V and $|M^*(V,P)|$ the number of methods in V with a too generous access

[3] http://pmd.sourceforge.net
[4] http://findbugs.sourceforge.net
[5] http://netbeans.org

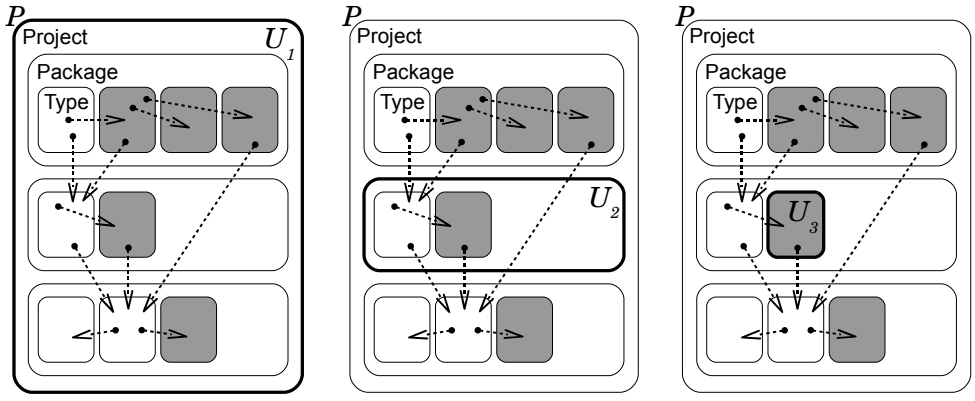

Figure 1. IGAT calculation

modifier, determined by the actual usage inside P. In cases with no method declarations in V, $IGAM(V, P)$ is defined as 0.

B. AccessAnalysis

AccessAnalysis is a free plug-in for the Eclipse IDE that determines minimal access modifiers for types and methods in Java projects. Based on the analysis results, it calculates the metrics IGAT and IGAM for each project, source folder, package, type, and method (for methods only IGAM).

The analysis is started manually by selecting one or more Java projects in Eclipse and clicking on menu entry "Analyse Access", which is added by the plug-in to the projects' context menu. Since the metrics are functions of the analysis context (P in Definition 1 and 2), it might lead to different results if several projects are analysed all together or one by one, provided there are dependencies across these projects. For a successful analysis, the selected projects must not have any compiler errors. After a successful run, the analysis results are shown in a tree-like table (Fig. 2 and Table 1). 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 member types can also be `private` or `protected`. Local and anonymous classes [2, § 14.3, 15.9.5] are not displayed.

C. Limitations

AccessAnalysis does not consider the accessibility of fields, because the common rule that these should always be `private` is easy enough and can already be checked by existing tools (e.g. PMD, FindBugs). To determine minimal access modifiers, AccessAnalysis uses static source analysis for finding every reference to a type or method. This works only 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 [8] [9], AccessAnalysis does not consider reflection, beside one exception: test classes and methods called by the *JUnit framework*[6]. Since AccessAnalysis analyses only plain Java Source Code, further code documents such as Java Server Pages (JSP) or framework configuration files are ignored.

D. Configuration

The JUnit test framework is very common and probably used by almost every Java project. JUnit test classes and methods have 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 that their `public` access modifier might be considered as too generous by following the rules of minimal access. Therefore AccessAnalysis contains 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 and methods.

[6] http://junit.org

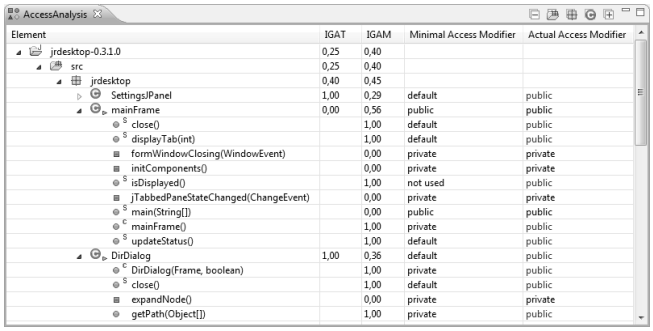

Figure 2. Display of results in AccessAnalysis

Level	IGAT	IGAM	Minimal Access Modifier	Actual Access Modifier
Projects	0..1	0..1	-	-
Folders	0..1	0..1	-	-
Packages	0..1	0..1	-	-
Types	0; 1	0..1	not used; `private`; default; `protected`; `public`	`private`; default; `protected`; `public`
Methods	-	0; 1	not used; `private`; default; `protected`; `public`	`private`; default; `protected`; `public`

Table I
VALUE SETS OF RESULT TABLE

Therefore custom *annotation types* [2, § 9.6] 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.

E. Evaluation

Using AccessAnalysis, we conducted a survey on twelve open source Java projects [10]. The results support our assumption that access modifiers are often chosen more generously than necessary. IGAT measures were around a mean of 0.32 and IGAM around 0.35. Especially top-level types were almost always declared as `public`, so that package interfaces typically expose more types than necessary. Only 2% of all investigated top-level types were encapsulated inside their package.

To give an impression of the analysis' performance, for one project with 1,104 classes it took around half a minute on our experimentation laptop (Dell Latitude E6420, Intel Core i5-2520M CPU @ 2.50GHz, 8 GB RAM, Solid State Disc, Windows 7 64 Bit, Eclipse Indigo).

III. TOOL ARCHITECTURE

The analysis contains two phases. The first and greater part is a complete source code analysis using the build-in Java DOM/AST API of Eclipse [11]. Here AccessAnalysis fills four data structures by processing the *abstract syntax tree* (AST) of every *compilation unit* (*.java file) included in the selected projects. The ASTs are processed with the Visitor Pattern [12] as specified by the API. In the second stage the data is merged to the result tree that will be displayed to the user.

A. First Analysis Stage

In the first part of the analysis the following data structures are filled:

(1) *The result tree containing projects, source folders, packages, types, and methods.* The first three levels are gathered from the project structure in the IDE. The type and method nodes are added when the corresponding declaration is processed by the AST visitor. The actual result values (Table 1) are set to the nodes in the second analysis stage.

(2) *A table that maps types to their minimal access modifiers.* For every reference to a type identified in one of the ASTs the required access level is calculated. For example, if the current AST node contains a local variable declaration and the type of the variable is from the same package, the required access level for this type use is default. So, the minimal access modifier of this type will be set to default if the corresponding table entry is not yet equal or more generous then default. When the AST visitor comes to a type declaration that has no table entry so far, an initial entry is added. Normally the minimal access modifier is initialized with not used, but there are some exceptional cases (see Section III-C).

(3) *A table that maps methods to their minimal access modifiers.* This is filled in a similar way as the type table.

(4) *A table that maps overriding/hiding methods to overridden/hidden methods.* When a method declaration is processed by the AST visitor, the type hierarchy is searched for an overridden or hidden [2, § 8.4.8] method. If one is found and it comes from inside the analysed project(s), an entry is added to the table.

B. Second Analysis Stage

In the second analysis stage the collected data is merged and the individual result values are set to the result tree nodes. The actual access modifiers of types and methods are gathered from the corresponding DOM elements. The minimal access modifiers of types can now simply be taken from table (2). The minimal access modifier of a method is the more generous one of either the one from table (3) or the minimal access modifier of the overridden method, if there is one in table (4).

The IGAM value of a method is 1 if its own actual access modifier is more generous than its own minimal one, otherwise 0. The same applies to the IGAT value of types. That means that nested types have no effect on the IGAT value of their enclosing type. In the result tree, all types are on the same level. The metric values of higher level nodes (projects, source folders, packages and, for IGAM, types) are calculated from the results in their successor nodes.

C. Extension Point for Initialising Minimal Access Modifiers

When the AST visitor passes the declaration of a type or method for which no reference has been visited yet, the minimal access modifier is normally initialized with not used, but AccessAnalysis contains an *extension point* [13, p. 637–660] to manipulate the initial minimal access modifier. This mechanism is applied for three use cases:

- To implement basic language rules: (a) interface methods and methods that implement interface methods are always `public` [2, § 6.4.4], (b) main methods are always `public` [2, § 12.1.4], (c) methods that override or hide other methods cannot reduce their accessibility [2, § 8.4.8]. The latter applies only in cases where the overridden/hidden method comes from outside the analysed projects; otherwise the minimal access modifier is the one of the overridden/hidden method.
- To check whether a type or method needs to be `public` for JUnit or whether a preset annotation constraint applies to the type or method (see Section II-D).
- For future extensions, e.g., to inject the results of preliminary analyses of further documents, e.g. JSPs or framework configuration files.

IV. Analysis Details

The explained algorithm walks through the AST of every compilation unit in the analysed projects and gathers all use relations to the projects' types and methods. In the end, the most generous necessary access level of all references to a type or method defines its minimal access modifier.

A. Necessary Access Level

The necessary access level for a single use of a top-level type depends on the relation between using and used type. For member type and method use, the relation between using and declaring type is crucial. For method calls, also the static type of the expression at which the method is called plays a role.

Example: In the following code snippet, `A` is the declaring type, `B` the expression type, and `C` the using type of the method call in line 13. The actual runtime type of the expression (`AnySubtypeOfB`) is irrelevant for the method call.

```
1   class A {
2       void foo() {
3           ...
4       }
5   }
6
7   class B extends A {
8   }
9
10  class C {
11      void bar() {
12          B b = new AnySubtypeOfB();
13          b.foo();
14      }
15  }
```

The Java DOM/AST API of Eclipse provides almost every information that is needed to determine the necessary access level of a single use relation. For every use relation, AccessAnalysis checks the following conditions [2, § 6.6] for each access level from most restrictive (not used) to most generous (`public`). If one access level fits, it is the necessary one.

- not used if a type or method is used by itself (recursion). For method calls, using, declaring, and expression type have to be the same.
- `private` if a member type or method is used from inside its own top-level class. For method calls, using, declaring, and expression type have to come from the same top-level class.
- default if a type or method is used from inside its own package. For method calls, using, declaring, and expression type and, additionally, all types in the type hierarchy between declaring and expression type have to come from the same package.
- `protected` if a member type or method is used by a subclass of the declaring class. For method calls, the using type has to be a subtype of the declaring type. Additionally, for nonstatic method calls, the expression type has to be the using type or one of its subtypes.
- `public` if a type or method is used anywhere else.

B. Special Cases

Beside those basic rules, there are some special cases and pitfalls that require further source code investigation.

- Interface methods and methods implementing interface methods have to be `public`; main methods have to be `public`; overriding and hiding methods must not reduce accessibility (see Section III-C).
- If a class inherits a method from another class to implement a method declared in an interface, the method in the superclass needs to be `public`. This pattern, known as "Marriage of convenience" [1, p. 530], cannot be recognized ad hoc by simply processing the individual ASTs of the involved types.
- Implicit Constructor calls (`super()`, [2, § 8.8.7]) are not part of the source code, but require access rights and so have to be taken into account.

- The Eclipse API returns the expression type of a method call only for qualified calls. Unqualified method calls require further investigations. In such cases, the type of the implicit expression can be the calling class itself, an enclosing class, or the type of a static import.
- AccessAnalysis does not consider `import` statements [2, § 7.5] as type use, since unused imports are needless and used imports will be recognized anyway when the imported type is actually referenced in the following source code. This practice does not work for static member import, since Eclipse does not offer any information about the declaring type when the AST contains a reference to a static imported member. We have chosen a pragmatic way and consider every static member import as use of the declaring type.

V. Conclusion

We presented AccessAnalysis, a tool for measuring the accessibility of types and methods in Java. The measuring is based on the concept of minimal access modifiers. AccessAnalysis helps to monitor interface evolution on several levels. It is primarily useful for stand-alone applications where the whole source code is available for analysis. It is less useful in the development of libraries or frameworks with a substantial amount of types and methods that are dedicated to the use in third-party applications. However, AccessAnalysis even offers the possibility to handle dedicated API types and methods separately by marking them with custom annotations.

AccessAnalysis calculates two software metrics, IGAT and IGAM, that represent the deviation between actual and minimal access modifiers. These metrics can support refactoring by disclosing the hotspots of too wide open interfaces. AccessAnalysis does not minimize interfaces automatically by applying the collected minimal access modifiers, although this would be possible. But interface design is more than a question of minimization. In the end, a developer has to decide about the truly necessary access modifier. The minimal access modifier of an element might even exceed the appropriate access level, when an insufficient encapsulation is already exploited by clients. However, the minimal access modifier, determined by AccessAnalysis, can serve as a useful reference.

The static source code analysis is done with the build-in Java DOM/AST API of Eclipse. It turned out, that the API is appropriate for such an analysis and provides all information needed to determine minimal access modifiers. However, some special cases and pitfalls have to be considered. We have described these cases in this paper.

AccessAnalysis is open-source and available for free under conditions of the Eclipse Public License (EPL) [14]. It can be downloaded from http://accessanalysis.sourceforge.net.

References

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

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

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

[4] 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.

[5] 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

[6] 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.

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

[8] 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.

[9] 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.

[10] C. Zoller and Axel Schmolitzky, "Measuring inappropriate generosity with access modifiers in Java systems," in *Proc. of The Joint Conference of the 22nd International Workshop on Software Measurement and the 7th International Conference on Software Process and Product Measurement (IWSM/MENSURA 2012)*, Assisi, Italy, Oct. 17–19, 2012.

[11] T. Kuhn and O. Thomann. Abstract syntax tree. Eclipse Corner Articles. [Online]. Available: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison Wesley Longman, Inc., 1995.

[13] D. R. Eric Clayberg, *Eclipse Plug-ins*, 3rd ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2008.

[14] Eclipse public license - v 1.0. Eclipse Foundation. [Online]. Available: http://www.eclipse.org/legal/epl-v10.html