# Controlling Accessibility in Agile Projects with the Access Modifier Modifier

Philipp Bouillon
Tensegrity Software GmbH
Im Mediapark 6a
D-50670 Köln
Philipp.Bouillon@tensegrity.de

Eric Großkinsky
LG Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
egrosskinsky@online.de

Friedrich Steimann
LG Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

**Abstract**. Access modifiers like `public` and `private` let the programmer control the accessibility of class members. Restricted accessibility supports encapsulation, i.e., the hiding of implementation details behind the interface of a class. However, what is an implementation detail and what makes the interface of a class is often subject to change: especially in an agile setting (with absence of an upfront design dictating accessibility levels), the interface of a class evolves much like its implementation, settling only towards the finalization of a project. However, while insufficient accessibility is reported by the compiler, excessive accessibility is not, the effect being that massively refactored programs usually end up with larger interfaces than necessary. With our ACCESS MODIFIER MODIFIER tool, we allow programmers to increase and keep accessibility at higher levels during the development phase, and reduce it only once the required access has stabilized. Fixed design decisions (such as a published API) can be designated by corresponding annotations, making them immune to changes through our tool. Evaluating the ACCESS MODIFIER MODIFIER on a number of internal packages taken from the JAVA open source community, we found that accessibility was excessive in 32% of cases on average.

**Keywords**. *Design, Encapsulation, Agile software development, Refactoring, Tool support.*

## 1 Introduction

In languages like JAVA supporting information hiding [19] through access modifiers such as `private` and `public`, any attempt to reference an insufficiently accessible class member results in a compile-time error [9]. By contrast, excessive accessibility does not — instead, it is often tacitly assumed that higher than required accessibility, or even accessibility without any access from within the program, is granted intentionally, i.e., that it reflects the designed interface (or API) of a class. However, in programming practice, in agile settings especially, the design of interfaces changes as part of the refactoring of code, and changes to accessibility are driven by what is necessary rather than what is superfluous, usually leading to the phenomenon that high levels of accessibility that are no longer needed are nevertheless maintained. While this may seem an inexcusable lack of discipline, we conjecture that it is also due to a lack of information: the maintainer of a class is not necessarily aware of all its clients and the access they require, so that reducing accessibility is a trial and error process. What would be needed instead is some kind of warning indicating where accessibility is unnecessarily high.

In JAVA, access modifiers do not only control accessibility of program elements, they also contribute to the semantics of a program. In particular, accessibility has an effect on dynamic binding and also on static method resolution under overloading and hiding [9]. Seemingly innocuous changes of access modifiers may therefore silently change the meaning of a program, so that tools preventing such unintended changes should be highly welcome.

To address these problems, we have devised a new tool, called ACCESS MODIFIER MODIFIER (AMM), that helps the programmer control the accessibility of class members. We have implemented this tool as a plug-in to the ECLIPSE Java Development Tools (JDT), and evaluated it by automatically applying it to several internal packages of large and well-known projects. The tool and a brief guide to its use are available for download at `http://www.fernuni-hagen.de/ps/prjs/AMM2/`.

The contribution of this paper is fourfold:

1. We define the notions of sufficient and excessive accessibility of class members.
2. We investigate the conditions under which excessive accessibility levels can be changed without changing program semantics.
3. We describe the implementation of a tool that helps control the accessibility of class members.
4. We present empirical evidence that our tool support for accessibility changes, reductions especially, is useful in practice.

The remainder of this paper is organized as follows. In Section 2, we take a quick look at how programmers set accessibility of class members in practice, and derive from their behaviour desirable tool support. In the sections that follow, we address our four contribution listed above. We conclude by discussing our approach and by comparing it to related work.

## 2 Motivation

In JAVA, accessibility levels of class members control the interfaces of classes and packages. As long as the design has not stabilized, these interfaces are subject to change. Changes may require an increase in accessibility, or may allow a decrease; in the first case, action by the developer is mandatory (otherwise the program will not compile), in the second, it is optional. In both cases, the changes are initiated by adding or removing dependencies in some remote place (different class or package) and therefore require the parallel editing of two source code locations (files), disrupting the work flow in an untoward way.

Under these conditions, programmers usually experience a certain tension between the goals “ease of coding” and “achieving maximum encapsulation”. The poles of this tension are succinctly described by the following, opposing approaches:

1. *Create Publicly Privatize Later* This is the liberal approach pursued by developers striving for maximum flexibility during development: all methods of a class are

inherited to its subclasses and can be accessed freely by other classes. While this design may be viewed as needlessly bloating the interface of a class, it saves the developers from anticipating adequate accessibility levels. At the same time, public accessibility facilitates unit testing which, depending on the testing framework used, requires public access to the methods under test. On the other hand, without additional API documentation it leaves users of the so-designed classes ignorant of which methods they can rely on (the stable, or published [6] interface) and which are subject to change without notice. Privatizing members later is possible, but difficult (see below); in particular, care must be taken that this does not change program semantics (see Section 4).

2. *Create Privately Publish Later* This is the cautious approach pursued by programmers caring about information hiding, acknowledging the fact that "it's easy to widen accessibility, but more difficult to reduce the accessibility of a feature in working code." [2] It leaves no doubt concerning the required accessibility of a member: if it is higher than private, then this is so because it is actually needed, not because someone speculated that it might be needed in the future [2]. However, even these developers must face the fact that as the design changes, a member may be left with excessive accessibility, and the resulting problem is the same as that for the first approach. Also, when it turns out that the accessibility must be increased, they cannot be sure that this change does not affect program semantics.

Both approaches are extreme and in programming practice, a mixture of both will likely occur. However, each approach is representative of one practical programming problem that we would like to address with our AMM:

1. For creating publicly and privatizing later, an indication that informs the developer of member declarations whose accessibility level is higher than required by design during and in particular at the end of development would be helpful. This accessibility should be reducible to the minimum required level using a corresponding micro-refactoring offered to the programmer (in ECLIPSE in the guise of a so-called Quick Fix). In doing so, the tool should be aware of whether the reduction is at all possible without changing program semantics. It should also be able to accept that a given member is part of the API, so that accessibility should not be reduced, or only criticized as being too high.
2. For creating privately and publishing later, when access to a member with insufficient accessibility is needed, the developer should be allowed to increase its accessibility without having to switch to its declaration site. This is already possible in ECLIPSE, although only via the detour of using the inaccessible member, thus forcing a compile-time error which can then be removed via the offered Quick Fix increasing accessibility as required. However, it would be more convenient were the developer allowed to inspect the hidden members and select from those (in the same manner so-called Content Assist works for accessible members), having the selected member's accessibility adapted before using it.[1] Again, the AMM tool should guarantee that this does not change program semantics, this time by not offering members whose accessibility level must be maintained.

[1] To whom this appears as sabotaging the very purpose of modularization (or information hiding [19]), be reminded that the strategy is "create privately, publish later". In agile development, corresponding design changes are a matter of fact. Cf. Section 7.

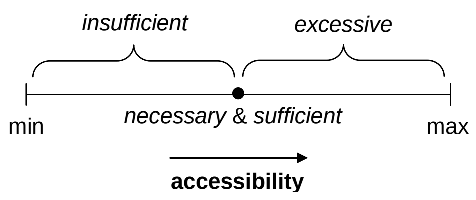


**Figure 1**. A class member's accessibility status is either *insufficient*, or *necessary and sufficient*, or *excessive*. While accessibility must be sufficient for a program to compile, excessive levels of accessibility are accepted. In absence of an API (i.e., for closed, monolithic programs), the necessary and sufficient accessibility level (depicted by the dot) of each program element is uniquely determined by the program and can be derived by means of a whole-program analysis.

The privatization functionality described above can also be useful when a project has been tested and is now to be released, so that the test suites are removed, allowing accessibility of methods previously required to be public solely for the sake of testing to be reduced. For this purpose, an automated execution of all suggested accessibility reductions is desirable.

## 3 Sufficient and Excessive Accessibility

Before elaborating on how the above goals can be achieved, an analysis of the problem is necessary. For this, we begin with a definition of the terms of sufficient and excessive accessibility.

*Sufficient* (or its converse, *insufficient*) and *necessary* (or its converse, *excessive*) accessibility are determined by the access control rules of the language, and by the mutual use dependencies of program elements organized in modules. JAVA has four accessibility levels named *public*, *protected*, *default* (deriving form the fact that it has no corresponding access modifier keyword; also called *package local*), and *private*, and two module constructs (module in the sense that something can be hidden inside it), namely *class* and *package*. Dependency is divided into *uses* (or *call*) dependency and *inheritance* (subclassing). Public accessibility of the members of a class lets them being depended upon by any other class, protected by all its subclasses and any class in the same package, default by any class in the same package, and private only by the owning class. These simple rules suffice to detect access violations, i.e., attempts to access class members declared to be out of reach (i.e., *inaccessible*, sometimes also be referred to as invisible, although this term is defined differently in the JAVA language specification [9]). If no access violations occur, the accessibility of all members is *sufficient*. It may however be *excessive*, namely when it is sufficient, but higher than necessary, so that accessibility of members can be reduced without becoming insufficient.[2] Figure 1 illustrates the situation.

To distinguish sufficient and excessive accessibility (defined by use) from the accessibility expressed by access modifiers (`public` etc., defined by declaration), we refer to the former as *status* and to the latter as *level* of accessibility (but omit this distinction if it is not clear from the context). The status of accessibility is a property of the level of accessibility and as such (indirectly) a property of a class member, even though insufficient accessibility is usually ascribed to concrete uses of a class member and not to the class member itself. Because in JAVA, the clients of a class member are not known to that member or its owning class (this is different, e.g., for EIFFEL, which has a dedicated export [17]), all accessibility states must be determined globally, i.e., by an analysis of the whole program.[3] There is however an important difference between the accessibility states *sufficient* and *excessive* with respect to the development process, i.e., with respect to adding and removing dependencies on class members during development:

- When adding a new or removing an existing dependency, (preservation of) *sufficient* accessibility can always be checked locally, i.e., solely by determining the position of the calling or subclassing class relative to the class depended upon in the package structure and in the class hierarchy.
- By contrast, *non-excessive* accessibility is harder to maintain. In particular, it cannot be checked locally: as shown in Figure 2, adding or removing a dependency may or may not affect the excessiveness of accessibility, and which is actually the case is not determined by the now using, or no longer using, class alone.

It is important to understand that accessibility *status* is determined by the actual use of (or dependency on) class members in a project, while accessibility *level* determines their actual usability. This inversion of relationship is reflected in the implementation of our AMM tool, which needs to set up inverse data structures (see Section 5).

## 3.1 Accessibility Status and APIs

Things are less clear-cut when projects are analysed that are designed for use by others or open for extension (such as libraries or frameworks). In such projects, necessary and sufficient accessibility levels are not only determined by the actual (present) dependencies, but also by the *designed interface* to the external world, which is often referred to as the *Application Programming Interface* (API). Members of the API may be required to be declared `public` even if not being accessed from within the project itself, or `protected` even without the existence of project-internal subclasses. In these

[2] In mathematical logic, a condition can be *necessary* (i.e., not *excessive*), but not *sufficient*. We do not consider this case here — a program with insufficient accessibility is incorrect (does not compile) and therefore outside the scope of this paper.

[3] Note that compilers commonly do not perform whole program analyses, but check sufficiency of accessibility per use. Insufficient accessibility is then marked as an error of the use site (attempted access violation) and not of the inaccessible class member. This reflects the viewpoint that the language should enforce information hiding. The scope of this paper is however slightly different.

```
package same;
class Called {
    private void m() {…} // insufficient accessibility
    public void n() {…}  // sufficient and excessive accessibility
    public void p() {…}  // necessary and sufficient accessibility
}

package same;
class Caller {
        …
        Called o = …
        o.m(); // compile-time error
        o.n(); // OK
        o.p(); // OK
        …
}

package other;
import same.Called;
class Caller {
        …
        Called o = …
        o.p(); // OK
        …
}
```

**Figure 2**. Insufficient, necessary and sufficient, and excessive accessibility of members of a serving class, determined by two client classes, one in the same, one in another package. When adding access to `n()` in `other.Caller`, it is not decidable locally whether its status of accessibility changes from *excessive* to *necessary and sufficient* (it does, actually). Vice versa, when removing access to `p()`, it is unclear whether status of `p()` changes from *necessary and sufficient* to *excessive* (it does again). Note that insufficient accessibility is usually marked at the call site (through a corresponding compile-time error), while excessive accessibility can only be ascribed to the called (or, rather, not called) site (cf. Footnote 3).

cases, accessibility of a class member is excessive only if the member is not part of the API, even if there is no actual dependency *within* the project requiring the given accessibility.

Unfortunately, JAVA has no linguistic means to express what is part of the API — the meaning of public or protected accessibility is unspecific in that use of the corresponding access modifiers cannot distinguish internally required from externally required accessibility [6]. It follows that if the API is not otherwise formally documented (e.g., through javadoc tags), it is impossible to decide mechanically whether a member's accessibility is excessive. To make up for JAVA's inability to distinguish API form internally required accessibility levels, some development teams have adopted the Eclipse Naming Conventions to designate packages whose public and protected methods are supposed to be *not* part of the API:

> *All packages that are part of the platform implementation but contain no API that should be exposed to ISVs* [Independent Software Vendors] *are considered internal implementation packages. All implementation packages should be flagged as internal, with the tag occurring just after the major package name. ISVs will be told that all packages marked internal are out of bounds. (A simple text search for ".internal." detects suspicious reference in source files; likewise, "/internal/" is suspicious in .class files).* [4]

and also:

> *Packages with* [the .internal] *prefix are implementation packages for use within the given module. Types and fields that are accessible within these packages MUST NOT be used outside the module itself. Some runtime environments may enforce this reduced accessibility scope.* [8]

Instead of marking packages as internal, we will introduce an annotation that tags class members as being part of the API, so that our tool never classifies their accessibility as excessive. However, we will make use of the "internal" naming convention in the evaluation of our approach (Section 6).

## 3.2 Accessibility Status and Subtyping

Subtyping dictates that accessibility of methods overridden in subtypes must not be lower than that in their supertypes (so that the cancellation of members by making them inaccessible is made impossible). While this is required by the principle of substitutability [16], it tends to get in the way when subclassing is done primarily for the sake of inheritance; often, then, many inherited class members are unneeded, so that their accessibility is factually excessive (and required only by the possibility of substitution, which may never take place). In this case, replacing inheritance with forwarding or delegation [12] is indicated.

However, it is possible that the necessary accessibility of a member overridden in a subclass is higher than that in its superclass. This is for instance the case when a client of the subclass requires public access, whereas for the superclass protected access suffices. When this additional client is moved to the same package as the subclass, the accessibility can be lowered to protected. If the accessibility required from the member in the superclass is then changed to default, the accessibility of both can be reduced simultaneously to this level.

Things are reversed when an increase of accessibility for a superclass's method is required. If its subclasses override the method with same (former) accessibility level, the increase in the superclass is not allowed unless the subclasses are changed with it. In these cases, increasing visibility of the whole hierarchy may be indicated.

Note that in JAVA, the same rules apply to static methods, for which hiding replaces overriding [9, § 8.4.8]: the access modifier of a hiding method must provide at least as much access as the hidden method, or a compile-time error occurs.

## 3.3 Interface Implementation

JAVA has a special type construct, named *interface*, whose members are implicitly public. Interfaces can be subtyped by classes, in which case the class must implement all methods declared by the interface. As with subclassing and overriding, visibility of implemented methods must not be reduced — in this case, it must remain public.

Things are slightly complicated by the fact that interface implementation introduces the possibility of multiple subtyping, i.e., that a class can have more than one direct supertype (a superclass and one or more interfaces). This leads to the constella-

```
class A {
    public void m() {…}
}

interface I {
    void m();
}

class B extends A implements I {
    …
}
```

**Figure 3**. Indirect accessibility constraint imposed by a subclass implementing an interface.

tion shown in Figure 3, in which visibility of m in A cannot be reduced, but not because A implements I, but because a subclass of it does.

### 3.4 Anonymous Subclasses

JAVA allows anonymous subclasses which can override methods just like ordinary subclasses. Their existence therefore has to be considered when searching for excessiveness of accessibility, and also when checking for a possible increase of accessibility in a superclass (see Section 3.2). A somewhat unexpected constraint results from anonymous subclasses *not* overriding methods: upon occurrence of the expression

```
new C(){}.m()
```

in a program, accessibility of m() in C cannot be reduced beyond what is required by the (location of the) expression.

## 4 Accessibility and Program Semantics

Access modifiers not only control the accessibility of class members by their clients, they also play a role when selecting from a number of competing member definitions. When the members are methods, this selection process is called binding. JAVA distinguishes static and dynamic binding; each will be considered separately.

### 4.1 Dynamic Binding: Overriding with Open Recursion

As discussed in Section 3.2, the accessibility of overriding and overridden methods depends on each other in that the accessibility of the overriding method in the subclass must be at least that of the overridden method in the superclass. One might be led to believe that accessibility of the methods can be changed freely as long as this constraint is satisfied. However, this is not the case.

```
class Super {
    public void m() {
        n();
    }

    public void n() {// private possible, but changes semantics
        System.out.println("Super.n");
    }
}

class Sub extends Super {
    public void n() {
        System.out.println("Sub.n");
    }
}

class Client {
    public static void main(String[] args) {
        Sub o = new Sub();
        o.m();
    }
}
```

**Figure 4**. Overriding and open recursion prohibiting reduction of accessibility: if accessibility of n() in Super is reduced to private, the overridden version is Sub is no longer called.

Subclassing comes with an interesting twist to uses-dependency: while methods defined in the superclass can be called from the subclass and its clients, they can also call methods defined in the subclass, namely via overriding and open recursion [20] (the mechanism behind the TEMPLATE METHOD pattern [7]). For overriding, the overridden method must be declared at least default if the subclass, the superclass and all intermediate classes are in the same package, and at least protected otherwise.

Now consider the code from Figure 4. It satisfies the condition that accessibility of n() is no less in Sub than it is in Super. Changing accessibility of n() in Super to private does not violate this condition. However, it changes program semantics: executed on an instance of class Sub, m() now calls n() in Super instead of in Sub — the dynamic binding has silently been removed. Vice versa, assuming that n() had been declared private in Super in the first place, increasing its accessibility to default, protected or public would also change semantics, this time by introducing a dynamic binding that was previously absent. In JAVA, access modifiers do not only decide over whether a program is correct (in that it respects the declared interfaces and thus information hiding), they also have an effect on program semantics.

With JAVA 5, this problem has partly been fixed by introducing the @Override annotation, which issues a compile-time error if a so-tagged method does not actually override. So if in Figure 4, n() in Sub were tagged with @Override, accessibility of n() in Super could not be lowered to the degree that it is no longer overridden (here private). The problem would have been completely fixed if the JAVA 5 compiler had required all overriding methods to be so tagged: in this case, increasing accessibility of n() in Super from private to some higher level would reject the definition of n() in Sub as lacking the @Override annotation. Note that C# has a required keyword override for overriding methods, which fully solves the problem; for JAVA, the ECLIPSE JDT offer a compiler switch whose selection makes the @Override annotation mandatory

```
class Super {…}

class Sub extends Super {…}

class Server {

    public void m(Super o) {
        System.out.println("m(Super)");
    }

    public void m(Sub o) {// private possible, but changes semantics
        System.out.println("m(Sub)");
    }
}

class Client {
    void callMSub() {
        Sub o = new Sub();
        Server s = new Server();
        s.m(o);
    }
}
```

**Figure 5**. Overloading prohibiting adjustment of accessibility: if accessibility of m(Sub) is reduced to private, m(Super) is called instead.

for overriding methods. In absence of either, an additional constraint for changing accessibility of methods is that it must not introduce or remove a dynamic binding.[4]

### 4.2 Static Binding: Overloading and the Most Specific Method

A second problem that is related to subtyping, but does not involve dynamic binding, occurs when determining the most specific method in a set of overloaded method declarations. In JAVA, when a method call matches several of a set of overloaded methods (i.e., methods available for the same class or interface with identical name and arity, but different parameter types), the most specific one is chosen ([9], § 15.12.2). If this most specific method is made inaccessible by reducing its accessibility, no compile-time error will result if a uniquely determined, less specific one can be linked instead. However, this static binding to a different method also changes the semantics of the program; Figure 5 gives an example of the problem.

The problem is somewhat worsened by the fact that as of JAVA 5, primitive and their wrapper types are mutually assignment compatible. The details of the resulting problems are discussed in [10]. Issues around variable parameter numbers (a new feature of JAVA 5) are not considered, neither here nor in [10].

A rather unexpected problem may occur when an up cast is used to disambiguate an otherwise ambiguous method call. If accessibility of the method that causes the ambiguity is reduced, the type cast becomes unnecessary, which may lead to a compile-time warning, or even error, with certain compilers.

[4] Unlike in [12], we do not check here for actual occurrence of open recursion, but only for presence of overriding. This makes the required analysis significantly simpler.

## 5 The Access Modifier Modifier Tool

To make the support described in Section 2 under the conditions of Sections 3 and 4 available to the programmer, we have implemented the AMM as a plug-in to ECLIPSE's JAVA Development Tools (JDT). The plug-in implements the JDT's builder extension point and is activated whenever the JDT have performed their own build.

The AMM integrates seamlessly with the JDT's standard user interface in that it

- adds a new type of warning equipped with two possible Quick Fixes (one for reducing accessibility, one for introducing an @API annotation) and in that it
- extends the existing Content Assist with the possibility to show as yet inaccessible members, associating an automatic increase of accessibility with their selection.

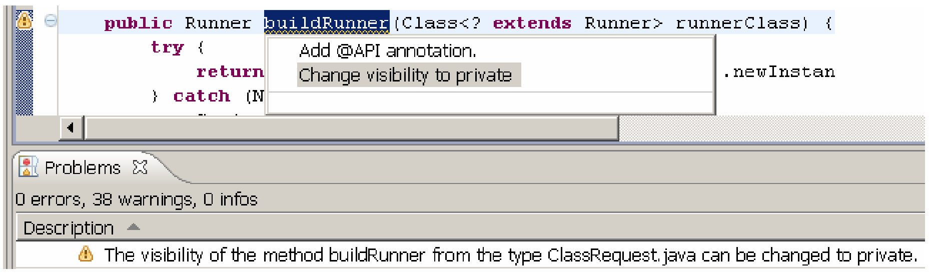
However, despite the possibility to reuse the user interface, the actual implementation of the tool meant considerable work from scratch. The reasons for this must be sought among the following.

When a class is compiled by the JDT, the compiler checks all references to other classes' members for their accessibility. If a member is inaccessible from the class, the corresponding reference (not the member declaration!) is marked as illegal. This can be done in a local manner, i.e., without considering other than the referencing and the referenced class (plus perhaps its superclasses). Sufficient accessibility of a referenced member (as its accessibility status; cf. Section 3) can be deduced from a compiling program, i.e., from a program that contains no illegal references to that member. Also, upon change of a reference (by adding, removing, or moving it to a different location) sufficiency of accessibility of a class member can be updated based on the change alone, irrespective of all other references: higher demands will lead to an error), while lower demands leave sufficiency untouched.

Things are quite different for the accessibility status taken care of by the AMM, namely excessive accessibility. From a legal reference to another class's member, or even from the fact that all references are legal with respect to the accessibility rules of JAVA, it cannot be deduced whether accessibility status is necessary and sufficient, or excessive (cf. Figure 1). Even worse, when a reference is added, removed, or moved to a different location, the change's effect on accessibility status is unpredictable without knowing the requirements of all other references. The AMM tool therefore requires significant additional data structures.

### 5.1 Full Build

For a full build, the AMM traverses the abstract syntax tree (AST) of all open projects in the workspace that depend on the project for which accessibility is to be controlled (and for which the full build has been triggered), and collects all calls to methods of this project. The set of methods can be further restricted by setting a corresponding package filter; such a restriction, for instance to internal packages (cf. Section 3.1), is often useful and has in fact been exploited in our evaluation of the AMM (Section 6). The call dependencies collected in this manner are stored in two hash tables, one — named *Callees* — keeping the set of methods called by each method, the other — named *Callers* — the set of types from which each method is called. The necessary and sufficient accessibility level can then be computed for each method as the maxi-



```
public Runner buildRunner(Class<? extends Runner> runnerClass) {
    try {
        return                                  .newInstan
    } catch (N
```

```
0 errors, 38 warnings, 0 infos
The visibility of the method buildRunner from the type ClassRequest.java can be changed to private.
```

**Figure 6**. Warning issued by the AMM tool, and quick fixes offered.

mum accessibility level required from the types stored in *Callers*, subject to the restrictions discussed in Sections 3 and 4, which are checked as described in [10]. After the build has completed and the computed warnings have been issued and the corresponding markers have been set (see Figure 6), the hash tables are stored on disk for later use by incremental builds.

### 5.2 Incremental Build

An incremental build is triggered by a change in one compilation unit (CU). The JDT notify the AMM builder of this change, who can then commence its action.

In a first step, the AMM builder updates the callees (as stored in *Callee*) of all methods contained in the changed CU (including added and deleted methods), and also the calling types of each newly called or no longer called method (as stored in *Callers*). In the second step, it visits all updated callee methods determined in the first step. Each of these methods is looked up in *Callers* to compute the new necessary and sufficient accessibility (again as the maximum of the accessibility levels as required by each calling type). The AMM builder then proceeds checking the preconditions for changeability of accessibility as described above, and updates warnings and markers correspondingly.

### 5.3 Reducing Accessibility or Adding an @API Annotation

The warnings computed by the AMM tool are displayed in the JDT's problem view, as shown in Figure 6. Corresponding Quick Fixes allow the reduction of visibility without opening the editor of the class hosting the member in question, or alternatively the insertion of an @API annotation. A second Quick Fix offered in the Problems view reduces accessibility according to all selected warnings at once (not shown).

**Table 1**. Projects used for evaluating our AMM tool (size is number of compilation units)

| PROJECT | FULL NAME AND VERSION | SIZE | SOURCE | ANALYZED PACKAGES |
|---|---|---|---|---|
| JUNIT | version 4.4 | 226 | junit.org | .internal* |
| SVNKIT | version 1.1.4 | 687 | svnkit.com | org.smatesoft.svn.core.internal.* |
| ECLIPSE | JDT core[§] version 3.3.1.1 | 1132 | eclipse.org | .jdt.internal.* |
| HARMONY | Apache Harmony JDK version 5.0 | 6765 | harmony.apache.org | *.internal* w/o *.test* and *.nls |

[§] Size includes all JDT projects of Eclipse; accessibility reductions have been computed for the `org.eclipse.jdt.core` project only.

## 5.4 Increasing Accessibility

Increasing accessibility with the AMM tool is integrated into the JDT's Content Assist, by providing a plug-in for the corresponding extension point. This plug-in adds to the standard Content Assist a page containing the members currently inaccessible from the object reference on which it is invoked. Upon selection of the desired member, the required increase of accessibility is checked against the preconditions listed in Sections 3 and 4 and, if satisfied, the increase is performed. The selected member is then automatically inserted in the code (the standard behaviour of Content Assist).

## 6 Evaluation

Designed to improve the consistency of source code, our AMM tool is not of the breed that is indispensable for the on-time delivery of correct programs. In order for it to be used, the imposed costs have to be carefully weighed against the expected benefits. The following evaluation should provide a basis for such a trade-off.

### 6.1 Usefulness

We have evaluated the usefulness of our AMM tool by applying it to internal packages of several open source JAVA programs. Programs were selected based on the existence of packages clearly designated as internal[5] and on the fact that the packages and their enclosing projects were extensively covered by JUNIT tests. We collected the numbers of possible reductions in accessibility indicated by the AMM tool, and checked unchanged semantics of the packages and their dependents by executing the unit tests after all suggested changes had been performed. The selected packages and size of containing programs are listed in Table 1; the results of the evaluation are given in Table 2.

[5] This turned out to be very selective — only few projects actually do this.

**Table 2.** Results of the evaluation (see text)

| PROJECT | MEMBERS OF INTERNAL PACKAGES | CHANGES FROM TO | | | | | | TOTAL | GAIN |
|---|---|---|---|---|---|---|---|---|---|
| | | public | | | protected | | default | | |
| | | protected | default | private | default | private | private | | |
| JUNIT | 131 | 2 | 8 | 9 | 1 | 15 | 4 | 37 | 28% |
| SVNKIT | 3555 | 11 | 731 | 219 | 214 | 60 | 2 | 1237 | 35% |
| ECLIPSE | 12780 | 431 | 1764 | 631 | 800 | 462 | 161 | 4249 | 33% |
| HARMONY | 1805 | 25 | 199 | 67 | 12 | 9 | 17 | 330 | 18% |
| total | 18271 | 470 | 2703 | 931 | 1027 | 546 | 184 | 5853 | 32% |

Overall, the relatively high numbers of possible reductions came as a surprise: although we had hoped that our evaluation would demonstrate the usefulness of the AMM tool, we did not expect unnecessary openness of internal packages to be that high. On average, 32% of all access modifiers of methods in internal packages were higher than required by their use of other packages. In particular, the relatively high number of unnecessary public accessibilities (70% of all excessive accessibilities) appears troubling.

There are various possible explanations for this. One is that access to classes hosting the members in question is limited, eliminating the need for individual class member access restriction. Another explanation is that internal packages have been designed with future internal extensions in mind, so that members are intentionally made accessible to other internal or non-internal project packages without already being used by these. One indication for this is the relatively high number of protected members that could have been declared `private`; for instance, these make up for 41% of all possible reductions in the internal packages of JUNIT. If this is actually the case, and if the authors insist on offering this internal interface to future extensions, introduction of a corresponding annotation would be in place (but see Section 7 for a discussion why we believe this is not necessary). A third explanation is that the internal naming conventions described in Section 3.1 are not strictly adhered to. One indication for this is that we found "external API" comments for methods in the internal packages of ECLIPSE's JDT. The last explanation is that developers have been uncertain about the accessibility status of their class members, and left accessibility on a level "that worked"; this is where our AMM tool steps in.

The findings delivered by our AMM tool are not without systematic errors. First, since its program analysis does not cover reflection, it is unable to detect and consider reflective calls. In the case of JUNIT this introduced two erroneous accessibility reductions (both from public to default). Generally, since JUNIT calls the methods representing test cases reflectively, test packages should be excluded from accessibility reductions. Note that this could be achieved automatically if the AMM treated the `@Test` annotations of JUNIT 4 like `@API` annotations and so left test cases untouched.[6]

[6] A similar problem occurred during testing the correctness of the AMM tool on the JDT core: its test methods call some of the tested methods reflectively, which is not discovered by the program analysis, thus causing failures. We therefore supplemented correctness tests of our implementation using unit tests from other projects without such problems (but which did not have packages designated as internal, which is why they were not included in our study).

**Table 3**. Spatial and temporal requirements of our AMM tool

| PROJECT | NO OF CALLERS[$] | SPACE (MB) MAX | SPACE (MB) RESIDUAL[§] | TIME (SEC)[*] FULL AMM BUILD |
|---|---|---|---|---|
| JUNIT | 45 | 167 | 0.45 | 4.5 |
| SVNKIT | 1735 | 231 | 4.83 | 100 |
| ECLIPSE | 5225 | 349 | 53.3 | 1,978 |
| HARMONY | 125 | 493 | 34.3 | 2,720 |

[$] number of entries in the *Callers* hash table (see Section 5 )
[§] as stored on disk
[*] obtained on a contemporary PC with dual core CPU rated at 3 GHz

Second, possible reductions of groups of methods (a method and its overridings; cf. Section 3.2) are not determined: if a method declared `public` that can be reduced to `protected` or default is overridden, the overriding methods with accessibility levels enforced by subtyping are not at the same time marked as reducible (because with the superclass's accessibility as is, they cannot). However, as soon as the accessibility of the overridden method is reduced, the overriding methods will be marked by the AMM as reducible. The so-induced reducibilities are immediately detected by the incremental build process; they have been included in the numbers of Table 2.

Last but not least, the JUNIT test coverage we required introduced a certain systematic error, since it may be the case that members have been made accessible only for the sake of testing, not because they are part of the designed API (cf. Section 2). However, this error does not enhance our results — rather, without the unit tests the overall sufficient accessibility could be even lower than what we are presenting here.[7]

### 6.2 Cost

Checking for excessive accessibility is not free, and if it is too expensive (in terms of time or memory required), it will likely be of no real use. We have therefore measured the requirements of our AMM tool; the results are compiled in Table 3. Note that memory demands must be divided into what is needed for the actual analysis and what is needed (as bookkeeping, or caching) for incremental builds; we have therefore determined the maximum memory requirement during a full build and the residual (after the build completed) separately. The maximum memory requirements for incremental builds during their computation are always lower than those for the full build. We did not attempt to measure incremental build time systematically, but our experiments showed that it is tolerable in most cases.

Clearly, time and space requirements of our current implementation for full builds are considerable (and unacceptable for large projects such as HARMONY). While we expect that a lot can be gained from optimization (which we have not attempted so far), it is also clear that a lot of overhead is imposed by ECLIPSE's builder interface, which required us to use explicit searches for types and methods, both imposing heavy performance penalties. Integrating the AMM into ECLIPSE's native JAVA build

[7] On the other hand, in absence of `@API` annotations, unit tests may be considered as simulating use by other clients.

process (so that it has access to the resolved parse tree) should speed it up considerably.

The long time required for a full build of HARMONY (especially when considering the comparatively small number of callers) must be ascribed to the collection and analysis of uses of the class `Object` and other frequently extended classes (HARMONY is a reimplementation of SUN's JDK): numbers of overridings of methods of these classes are literally in the thousands, so that precondition checking (which involves a detection of overriding) using the JDT's search facilities (cf. above) takes its toll.

## 7 Discussion

For the first version of our AMM tool (described in detail in [10]), we assumed a much more process-oriented viewpoint and designed a system in which accessibility levels could be changed collaboratively. According to this viewpoint, a developer could "open" a class (i.e., all its members) for maximum accessibility, use the members as desired, and later "close" it to the original accessibility levels. Openings and closures could be stacked, and all changes were stored in annotations parameterized with the author and time of action. This documented changed interface designs automatically, and always allowed a rollback to the original design. By contrast, the current version of the AMM tool does not record the accessibility levels it overrides, so that in case a change in design is changed back, the original accessibility levels (and the resulting interfaces) are unknown. However, assuming that accessibility was at the lowest possible level (interfaces were minimal) prior to the first design change, this is no problem, since the original level follows from the original design: reverting to the original design lets the AMM tool compute the original accessibility levels (unless `@API` annotations have been added or removed).

When we first devised the new AMM tool, we called the `@API` annotation `@Sic` (Latin for "so", meaning "so intended"). Technically the same, `@Sic` was more neutral with respect to intent, i.e., its only expressed purpose was that the so tagged access modifier should not be changed by the AMM. Therefore, `@Sic` could also rightfully be used for designating internal interfaces, in particular for members whose accessibility is higher than currently required by the project itself *or* its API, to indicate that the project has been prepared for *internal* extension (for instance in future releases; cf. the discussion of our findings in the case of JUNIT in Section 6.1). However, we maintain that this would taint the annotation with problems it is trying to avoid: as design changes, `@Sic` annotations must be added and removed, and if the latter is forgotten, accessibility will become excessive again.[8]

[8] One could argue that the same is true for the `@API` annotation; however, external interfaces are more prominent than internal ones and thus also more carefully maintained. In particular, the `@API` annotation can be used for other purposes as well, for instance for the generation of documentation. If by all means desired, an `@II` annotation could be added to designate internal interfaces.

## 7.1 Related work

Although clearly a practical problem, dealing with accessibility seems to have attracted not much attention from the scientific community. One of the few exceptions we are aware of is the work by Ardourel and Huchard [1], whose notion of access graphs helps with finding fitting access levels on a more general, language-independent level. Based on these access graphs, the authors offer various tools for extracting, displaying, analysing, and editing access information, and also for generating the access modifiers for programs in a specific language. However, the functionality of our AMM tool seems to have not been implemented.

Deriving necessary and sufficient accessibility from a program is related to, but sufficiently different from, type inference [11, 18, 24]. In fact, languages like JAVA mix type and accessibility information: while a type usually restricts the possible values of typed expressions and with it the operations that can be performed on them (or, rather, on the objects they deliver), accessibility restricts the set of possible operations (but not the values!) depending on where the expression occurs relative to the definition of the type. Thus, objects of the same type, or even the same object, may appear to have different capabilities depending on by whom they are referenced, even if the type of the reference is the same. This is orthogonal to access control through differently typed references (polymorphism): in JAVA, this would likely occur through context-specific interfaces [21, 23].

Our work must not be confused with that on *access rights analysis* as for instance performed by Larry Koved and co-workers [14, 15]. In JAVA, access rights (which are a different concept than access control [9]) are granted by existence of permission objects, instances of subclasses of the special class `Permission`. The required access rights cannot be checked statically, but must be computed using an analysis of control and data flow. Irrespective of all differences, adequately setting access rights suffers from the same problem of finding out what is necessary and sufficient: while insufficient access rights can be identified during testing (there is no static checking of access rights), there is no guarantee that these satisfy the *principle of least privilege*. Koved et al. have solved the problem using a context-sensitive, flow-sensitive interprocedural data flow analysis; as it turns out, their precision is much higher than can be made use of by the JAVA security system with its relatively coarse granularity. However, none of the results can be transferred to our approach, since JAVA's access control is static, ignoring all data flow.

The AMM can be viewed as a refactoring tool with built-in smell detector [5]. While the refactoring itself is trivial, smell detection and checking of preconditions (i.e., excluding occasions in which changing accessibility would change program semantics) is not; in particular, both require efficient implementations for the tool to be usable.

In response to strong criticism of the aspect-oriented programming community's disrespect of traditional interfaces and modularization (see e.g. [22] for an overview), it has been suggested that aspect-related interfaces are computed only once the system has been composed [13]. In a way, our AMM tool is capable of doing precisely this for traditional (i.e., non-aspect related) module (class) interfaces: if the *Create Publicly, Privatize Later* (Section 2) approach is pursued, the project can start with no interfaces at all, and the AMM can compute them once the project is completed. However, this approach is only feasible if there is no a priori design, especially if there is no need for a design that allows distribution of modules to different teams so that these can work independently. The latter was of course Parnas's original motivation behind the conception of information hiding and modularity [19] — in fact, he makes no secret of his opinion that language designers misunderstood what modularity is all about [3]. Leaving internal control of accessibility to our AMM tool and using `@API` annotations for the published interfaces separates language issues from the designers' intent.

### 7.2 Future Work

There are several things we did not consider in this paper and the current implementation of the AMM. First and most importantly, we did not consider field members, even though their accessibility states are interesting for the very same reasons as those of methods. This decision was driven by our more general interest in interfaced-based programming [21, 23], and the fact that interfaces do not publish field members. However, consideration of field access should pose no theoretical problems, although it means considerable extra work (due to different rules for accessibility under inheritance).

Second, accessibility of classes could also be controlled through the AMM. Again, we did not pursue this, which saved us from having to deal with a certain combinatorial complexity: if all members of a class drop below certain accessibility, and if there exist no references to the class itself requiring higher accessibility, would it make more sense to lower the accessibility of the class instead of that of its members?

Last but not least, it is tempting to try and re-implement the AMM using a constraint satisfaction framework such as the ones described in [11, 24]. Besides being a theoretical challenge (how does necessary and sufficient accessibility relate to an inferred type?), it should be interesting to see if constraint solution strategies exist that outperform our conventional implementation described in Section 5.

## 8 Conclusion

Evidence we have collected suggests that even in well-designed JAVA projects, accessibility of class members often exceeds what is required by the access rules of the language, or dictated by the API of the projects. Convinced that finding out and setting the minimum required access level of a class member is a real problem for the programmer, we have devised a tool that does this completely automatically. The tool is based on a whole-program analysis which it performs and maintains as part of the project build process, thereby allowing complete and omniscient control of accessibility. The costs associated with this tool, at least as it is currently implemented, are non-negligible.

## References

1. G Ardourel, M Huchard "Access graphs: Another view on static access control for a better understanding and use" *Journal of Object Technology* 1:5 (2002) 95–116.
2. *Create Privately Publish Later* (http://c2.com/ppr/wiki/JavaIdioms/CreatePrivatelyPublishLater.html).
3. PT Devanbu, B Balzer, DS Batory, G Kiczales, J Launchbury, DL Parnas, PL Tarr "Modularity in the new millenium: A panel summary" in: *ICSE* (2003) 723–724.
4. *Eclipse Naming Conventions* (http://wiki.eclipse.org/Naming_Conventions#Internal_Implementation_Packages).
5. M Fowler Refactoring: Improving the Design of Existing Code (Addison-Wesley 1999).
6. M Fowler "Public versus published interfaces" *IEEE Software* 19:2 (2002) 18–19.
7. E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns – Elements of Reusable Software* (Addison-Wesley, 1995).
8. *Package Naming Conventions Used in the Apache Harmony Class Library* (http://harmony.apache.org/subcomponents/classlibrary/pkgnaming.html).
9. J Gosling, B Joy, G Steele, G Bracha *The Java Language Specification* (http://java.sun.com/docs/books/jls/).
10. E Großkinsky *Access Modifier Modifier: Ein Werkzeug zur Einstellung der Sichtbarkeit in Java-Programmen* (Master-Arbeit, Lehrgebiet Programmiersysteme, Fernuniversität in Hagen, 2007).
11. H Kegel *Constraint-basierte Typinferenz für Java 5* (Diplomarbeit, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen 2007).
12. H Kegel, F Steimann "Systematically refactoring inheritance to delegation in Java" in: *ICSE* (2008).
13. G Kiczales, M Mezini "Aspect-oriented programming and modular reasoning" in: *ICSE* (2005) 49–58.
14. L Koved, M Pistoia, A Kershenbaum "Access rights analysis for Java" in: *OOPSLA* (2002) 359–372.
15. G Leeman, A Kershenbaum, L Koved, D Reimer "Detecting unwanted synchronization in Java programs" in: *Conf. on Software Engineering and Applications* (2004) 122–132.
16. B Liskov, JM Wing "A behavioral notion of subtyping" *ACM Trans. Program. Lang. Syst.* 16:6 (1994) 1811–1841.
17. B Meyer *Object-Oriented Software Construction* 2nd edition (Prentice Hall International, 1997).
18. J Palsberg, MI Schwartzbach "Object-oriented type inference" in: *Proc. of OOPSLA* (1991) 146–161.
19. DL Parnas "On the criteria to be used in decomposing systems into modules" *Commun. ACM* 15:12 (1972) 1053–1058.
20. BC Pierce Types and Programming Languages (MIT Press 2002).
21. F Steimann, P Mayer "Patterns of interface-based programming" *Journal of Object Technology* 4:5 (2005) 75–94.
22. F Steimann: "The paradoxical success of aspect-oriented programming" in: *OOPSLA* (2006) 481–497.
23. F Steimann "The Infer Type refactoring and its use for interface-based programming" *Journal of Object Technology* 6:2 (2007) 67–89.
24. F Tip "Refactoring using type constraints" in: *SAS* (2007) 1–17.