From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility

Friedrich Steimann and Andreas Thies

Lehrgebiet Programmiersysteme
Fakultät für Mathematik und Informatik
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org, Andreas.Thies@fernuni-hagen.de

Abstract. Contemporary refactoring tools for JAVA aiding in the restructuring of programs have problems with respecting access modifiers such as `public` and `private`: while some tools provide hints that referenced elements may become inaccessible due to the intended restructuring, none we have tested prevent changes that alter the meaning of a program, and none take steps that counteract such alterations. To address these problems, we formalize accessibility in JAVA as a set of constraint rules, and show how the constraints obtained from applying these rules to a program and an intended refactoring allow us to check the preconditions of the refactoring, as well as to compute the changes of access modifiers necessary to preserve the behaviour of the refactored program. We have implemented our framework as a proof of concept in ECLIPSE, and demonstrated how it improves applicability and success of an important refactoring in a number of sample programs. That our approach is not limited to JAVA is shown by comparison with the constraint rules for C# and EIFFEL.

"*Moving state and behavior between classes is the very essence of refactoring.*" [4]

1 Introduction

In object-oriented programming languages like C++, JAVA, and C#, information hiding [17] is supported by access modifiers such as `public` and `private`. Their disciplined use contributes to modularization and, thus, the design of a program.

Refactorings change a program's design without altering its (externally visible) behaviour [4]. Insofar as the change affects the division of the program into modules, access modifiers must be updated during the refactoring process to reflect the new modularization. However, while insufficient accessibility is routinely reported by the compiler, excessive accessibility is usually not and therefore often forgotten [1]. Worse still, in JAVA the change of access modifiers can have an effect on static and dynamic binding, changing the meaning of a program [1, 14, 19].

Refactoring tools are metaprograms aiding the programmer in the often tedious and error-prone refactoring process. Contemporary IDEs such as ECLIPSE [3], NETBEANS [15], and INTELLIJ IDEA [9] come with various refactoring tools, usually including

support for renaming program elements, moving elements, and modifying the type hierarchy. However, as we will see, when it comes to maintaining accessibility most refactoring tools are flawed, not only in rare corner cases. As we will also see, the problem is not caused by negligence of the programmers who implemented the tools, but the tremendous complexity of the programming languages used today, and the myriad of different constructions they allow.

In this paper, we present a constraint-based approach to modelling the access control[1] rules of JAVA that makes it easy for a refactoring tool to respect them. In particular, we show how a *change of accessibility* of a declared entity, as well as how a *change of location* of a declared entity and its contained references, propagate through a program, and how these changes are constrained by references to the declared entities and by other declarations. This enables us to enhance important refactorings such as MOVE TYPE/MEMBER and PULL UP/PUSH DOWN MEMBER by adding necessary preconditions that are currently unconsidered, and also by adding mechanics enabling applications that currently lead to failure. Our approach is analogous to that taken by the type-related refactorings described in [22], but remains completely orthogonal in the problems it addresses. Also, our definition of foresight rules anticipating the changes performed by an intended refactoring appears to be novel.

The remainder of this paper is organized as follows. In Section 2, we motivate our work by presenting a number of basic problems current refactoring tools have, and by arguing why existing related work does not address them sufficiently. In Section 3, we develop our formal framework of accessibility constraints and present the constraint rules that model JAVA's access control. In Section 4 we show how these constraints and their generation integrate into the refactorings we aim to improve. Section 5 presents the implementation of our framework in ECLIPSE's JAVA DEVELOPMENT TOOLS (JDT) and shows how we have tested and evaluated it. Section 6 discusses our work, its limitations, and its potential for performing systematic programming language comparisons.

2 Motivation

2.1 Problems

Moving a class without adapting accessibility can break the code. For instance, moving class `B` in the JAVA program

```
package a;
class A {
  B b;
}

package a;
class B {}
```

to another package with the corresponding refactoring tools of ECLIPSE, NETBEANS and IDEA will produce a compilation error, since for the class `B` to be accessible from other packages, it needs to be declared `public`, which the tools ignore (only IDEA

[1] not to be confused with access rights [11]

issues a warning that B will become inaccessible for A). Note that this is not a problem of JAVA's language design, but a necessary consequence of modularization: access across packages should be restricted to elements declared public. Moving B therefore either breaks the designed modularization and should be prevented, or it constitutes a design change that should be reflected in a change of the corresponding access modifiers.

While the above problem is detected by the compiler and easily responded to, the situation becomes more complex when members of B are accessed. For instance, moving class B in

```
package a;
public class A {
  void n() { (new B()).m("abc"); }
}

package a;
public class B {
  public void m(Object o) {..}
  void m(String s) {..}
}
```

to another package will not produce a compilation error, but instead change the meaning of the program: rather than the method n in A calling m(String) in B as before the change, m(Object) gets invoked instead. The corresponding refactoring tool of ECLIPSE performs the change without warning; NETBEANS displays that B.m(String) is referenced and IDEA warns that it becomes inaccessible from A, but neither indicates that the refactoring will change the meaning of the program.

The change of meaning can be detected by observing that the static binding of the method call has changed. However, this alone is not sufficient, as the following example shows:

```
package a;
public class A {
  void m(String s) {..}
  void n() { ((A) new B()).m("abc"); }
}

package a;
public class B extends A {
  void m(String s) {..}
}
```

Again, moving B to another package changes the meaning of the program, yet this time not because the binding changes, but because m(String) in A changes its status from being overridden in B to not being overridden, so that calling m(String) on a receiver of static type A is no longer dispatched to the implementation in B. In ECLIPSE and NETBEANS, this change of meaning goes unnoticed, IDEA notes that class A contains a reference to class B, but this is not indicative of the problem.

While all the above sample problems can be easily fixed by adapting the accessibility of members to preserve program meaning, in real programs there may be ripple effects that are difficult to oversee, and also unobvious conditions that prevent such changes. For instance, if

```
package a;
public class C extends B {
  void m(String s) {..}
  public void m(Object o) {..}
}
```

is added to the previous example, the accessibility of m(String) in B cannot be increased to public without also increasing its accessibility in C. However, increasing accessibility of m(String) in C may be contraindicated, as it can change meaning of another call:

```
package b;
class D {
  void n() { (new C()).m("abc"); }
}
```

Although all three IDEs offer a refactoring for changing accessibility of methods (as part of changing their signature), none of them notes the change of binding this entails.

It should be clear from these examples that for larger programs, the situation quickly becomes unmanageable for a human programmer. Reliable tool support is therefore needed.

2.2 Related Work

That moving classes, fields, and methods of an object-oriented program can be a non-trivial problem was already recognized by Opdyke in his doctoral thesis [16]. However, despite a presentation of formal preconditions, these seem to be only loosely related to a concrete language (C++), and do not seem to be thoroughly checked for completeness. For instance, the preconditions for pulling up a member variable (field) state that "the variable is defined identically in all subclasses where it is defined" and that "the variable isn't already defined locally in (as a private member of) the superclass" [p. 73]. However, if one of the subclasses has another superclass with a variable of the same name (that was previously hidden), an ambiguity arises for accesses of the variable from the subclass (cf. Section 3.1, Inh-2). Also, Opdyke's treatment of access modifiers and how they are to be handled in refactorings is only cursory.

Contemporary refactoring tools such as those integrated in the ECLIPSE JDT [3], in NETBEANS [15], and in IDEA [9] all include some basic precondition checking (in IDEA including the issuing of warnings when a declared entity is moved out of reach), and some (notably IDEA) also present a list of references potentially directly affected by a refactoring, but none of them correctly predicts the change of semantics provoked by the examples of the previous subsection and the subsections that follow, and none offers a change of access modifiers that would prevent such changes or avoid compilation errors. The understanding of the consequences of such refactorings is therefore the duty of the programmer.

The problem of maintaining accessibility is related to, yet sufficiently different from, making sure that all bindings are preserved under the RENAME refactoring [18]. It is similar in that each reference must refer to the same declared entity before and after a refactoring (or otherwise the meaning of the program changes). It is different in that maintaining static binding alone is not enough (as the above example with the lost dynamic binding suggests), and that it is not achieved by changing references to a declared entity (by renaming them as well, or by adding necessary qualification [18]), but by changing the (accessibility of) the declared entity itself. Also, as the last of the above examples suggested, changes of accessibility may be constrained by the accessibility of other declared entities, so that the refactoring may have ripple effects. In-

version of the lookup of a declared entity as resorted to in [18] does not point to these indirect constraints and is therefore insufficient to solve our problem.

That reverse lookup is indeed insufficient became clear to us during the development of our ACCESS MODIFIER MODIFIER (AMM), a smell detection and refactoring tool that marks all methods with excessive accessibility and offers its reduction to the lowest level tolerated by the program [1]. The AMM maintains reverse lookup tables for every method, pointing from that method to its references. However, to deal with the binding problems sketched above, we had to implement additional lookups and checks reflecting the relevant rules of the language specification. Since the checks and lookups were hard-coded for a specific problem, namely the independent change of accessibility of a single method, retrofitting them to a different purpose (such as precondition checking for general MOVE refactorings), or even to a different target language, amounts to rewriting them. Because the problem itself seems rather general, we thought that a more generic, problem-independent formulation of the conditions under which accessibility could be changed would be desirable.

Such a formulation has been delivered as part of a formal model of JAVA written for the theorem prover ISABELLE/HOL [19]. Using this formalization, some interesting runtime properties of JAVA programs concerning access integrity could be shown. However, both model and theorem prover are rather heavy-weight and have to our knowledge not yet been utilized in refactoring tools.

A much lighter declarative approach to controlling access has been pursued in KACHEK/J, a tool that infers object encapsulation properties for JAVA programs [7]. KACHEK/J uses constraints to express a set of rules that allow the inference of confinement, i.e., that no aliases to instances of a confined type exist outside its defining package. The constraints basically make sure that confined types are neither declared public nor cast to non-confined supertypes, that they cannot be the types of public or protected members, and that methods inherited by them cannot leak aliases to the this pointer. While these constraints add confinement as a new property to the language (rather than model existing ones, as we intend), to improve this property in existing programs the first author of [7] has developed the JAVA ACCESS MODIFIER INFERENCE TOOL (JAMIT) [7, 8], which also builds on constraints. However, the constraints of JAMIT model only those aspects of JAVA access modifiers that are relevant to the virtual machine (JAMIT operates on byte code), and do not deal with possible changes of bindings that result from moving program elements.

3 Accessibility constraints

Following the approach of JAMIT [8], we model the access control rules of JAVA using constraints, making our above identified refactoring problems solvable by constraint programming. Constraint programming usually consist of two parts:

1. the generation of the constraints describing the problem, and
2. constraint satisfaction, i.e., the computation of a solution for the generated constraints.

Each generated constraint constrains one or more variables by setting up relations between them or assigning constant values to them. Through shared variables, the

generated constraints form a network, referred to as *constraint set* hereafter; the solution of a constraint set consists of assignments to the variables that satisfy all constraints. Generally, a constraint set can have arbitrarily many (including no) solutions; in case more than one solution exists, one is usually interested in one that satisfies certain additional conditions (not expressed as constraints). Although the solution of a constraint set is generally problem-independent, the additional conditions can lead to algorithms finding the best solution efficiently.

The constraints describing a particular refactoring problem are usually generated from the program to be refactored by applying a set of *constraint generating rules*, or *constraint rules* for short [22]. The variables in the generated constraints represent those parts of the program that can be changed to solve the problem. Constraint generation also assigns the variables of the constraints initial values; these values reflect the program as it is at the outset of the problem (when the constraints were generated), that is, before the refactoring is performed.

A constraint set generated from a (syntactically and semantically) correct program always has a solution, and in particular all constraints are satisfied by the initial variable assignments, or otherwise the constraint rules are inconsistent. Vice versa, any assignment to variables that solves the constraint set must represent a correct program, or otherwise the constraint rules are incomplete. Therefore, given a complete set of constraint rules, if another than the initial solution has been found, adapting the original program to the changed variable values (so that constraint generation would have extracted these values as initial had it been applied to the adapted program) will lead to a (syntactically and semantically) correct program. We refer to adapting the program so as to reflect the variable values of a new solution as *writing back the solution*.

3.1 Generating accessibility constraints

Basics For our purposes, an object-oriented program consists of a set, D, of declared entities [6, §6.1] d, d_1, etc., and a set, R, of references r, r_1, etc. referring to declared entities. The set of declared entities D is partitioned into a set of classes, C, a set of interfaces, I, a set of methods (including constructors), M, and a set of fields, F.[2] D also contains a subset of declared entities (including all constructors) declared as static, S. We express the binding of a reference r to a declared entity d by a function

$$\beta: R \rightarrow D \qquad \text{(binding)}$$

where $\beta(r) = d$ means that reference r binds to declared entity d. One common invariant of refactorings is that bindings are not changed.

A program is further divided into a set, L, of locations l, l_1, etc. Each $d \in D$ is declared, and each $r \in R$ resides, in a location $l \in L$.[3] In languages allowing nesting of declarations, the location is conveniently expressed by a path expression involving all

[2] The set of variables (temporaries and formal parameters) is not contained in D, since their access cannot be modified.

[3] The location of a declaration element is sometimes referred to as its declaration space [13], and is not to be confused with its scope.

containing declarations. To facilitate reading, we tag declared entities and references in the code we are referring to using comments, as in

```
package a; class /*d1*/A { /*r1*/B b; }
```

We refer to the location of so tagged entities and references by the function

$$\lambda: D \cup R \to L \qquad \text{(location)}$$

For instance, the location of d_1 (class A) in the above program, $\lambda(d_1)$, is a (the containing package) and that of r_1, $\lambda(r_1)$, is a.A (the containing class).

In JAVA the accessibility of a declared entity is determined by an access modifier preceding its declaration. We write $\langle d \rangle$ for the declared access modifier of d. The set of available access modifiers, A, is $\{public, protected, package, private\}$.[4] Its elements are totally ordered: $public > protected > package > private$, where $>$ means granting greater access. As usual, we write $\geq$ to denote greater than or equal.

Whether a reference can access a declared entity is determined by the access rules of the language. In order to maintain a certain language independence (and also because they are quite intricate in the case of JAVA), we model the access rules as a function

$$\alpha: L \times L \to A \qquad \text{(required access modifier)}$$

where the first argument is the location of the reference, the second is the location of the referenced declared entity, and where $\alpha(\lambda(r), \lambda(d))$ computes the smallest access modifier for d granting r access to d. α may be considered an inverse of the so-called *accessibility domain* [13], mapping a declared entity and its declared accessibility to all locations in the program text in which access to the member is permitted. We model α as a function of a pair of locations rather than of $R \times D$ since the access modifier required for accessibility does not depend on individual references or declared entities, but where they are located.[5] Also, as we will see, not the references or declared entities, but their locations are the variables of our constraints.

For a declared entity d that is a member of a type, the location of a reference r to d may be insufficient to determine d's accessibility — the (static) type through which d is accessed is also significant. We model this through a function

$$\rho: R \to L \qquad \text{(receiver)}$$

computing the location corresponding to the body of the receiver type. For instance, in the program

```
class A {
  B b = new B();
  int i;
  void m() { /*r1*/i = 1; /*r2*/b.i = 2; }
class B extends A {}
```

$\rho(r_1)$ evaluates to A and $\rho(r_2)$ evaluates to B.

We are now equipped to state our first constraint rule.

---

[4] Not every declaration element can use every access modifier — the domain of legal access modifiers depends on the kind of element that is declared, and where it is declared. As will be seen below, we model this as a constraint rule.

[5] The one exception, access to protected members, is modelled as a constraint rule (Acc-2).

Accessing Accessing a declared entity d via a reference r requires that the declared accessibility of d, $\langle d \rangle$ (its access modifier) is equal to or greater than the accessibility required by the language's access rules. To express this, we introduce the following constraint rule:

$$\beta(r) = d \quad \Rightarrow \quad \langle d \rangle \geq \alpha(\lambda(r), \lambda(d)) \qquad \textbf{(Acc-1)}$$

Applied to the JAVA program

```
package a; class A { /*r1*/ B b; }
package a; class /*d1*/ B {}
```

we obtain the constraint $\langle d_1 \rangle \geq \alpha(\lambda(r_1), \lambda(d_1))$. The variables of the constraint are:

- $\langle d_1 \rangle$, the declared access modifier of d_1,
- $\lambda(r_1)$, the location of r_1, and
- $\lambda(d_1)$, the location of d_1.[6]

As noted above, for the initial assignments of the variables derived from a syntactically and semantically correct program, constraints are always solved; note how this is indeed the case for the above example, in which $\langle d_1 \rangle = \alpha(\lambda(r_1), \lambda(d_1)) = \textit{package}$.

Now a refactoring may change the values of one or more variables, possibly violating the constraint. For instance, when class `A` is moved to another package, $\lambda(r_1)$ changes its value so that α evaluates to *public* and the constraint is no longer satisfied. To satisfy it, either the declared access modifier $\langle d_1 \rangle$ has to be changed to *public*, or class `B` has to be moved to the same location.[7] While the constraint itself is neutral to the chosen solution, the constraint satisfaction algorithm can be adapted to compute the one that is required (or makes most sense) for the given refactoring.

In the special case of protected access, it must be made sure that a "protected member or constructor of an object may be accessed from outside the package in which it is declared only by code that is responsible for the implementation of that object" [6, §6.2.2]. This is achieved by the additional constraint rule

$$\beta(r) = d \wedge \alpha(\lambda(r), \lambda(d)) = \textit{protected} \wedge d \notin S \wedge \rho(r) \notin \text{subclasses}(\lambda(r)) \cup \{\lambda(r)\} \quad \Rightarrow \quad \langle d \rangle = \textit{public} \qquad \textbf{(Acc-2)}$$

in which subclasses($\lambda(r)$) represents the union of the locations corresponding to the bodies of (true) subclasses of the class whose body corresponds to $\lambda(r)$. Note that Acc-2 does not replace Acc-1 in case of accessing protected members — it only adds a stronger constraint.

Inheritance JAVA's access rules require accessibility of an inherited member as if it were accessed as a member of the base class [6, 14, 19]. Therefore, Acc-1 covers access of inherited members as well. For instance, in

---

[6] Note that in JAVA, the default constructor of a class may be implicitly accessed by its subclasses' constructors. In these cases, corresponding constraints must be created without presence of explicit references.

[7] Note that if class `B` is moved first, the constraint generated from Acc-1 only requires that r_1 is also moved. Other rules of the language may require that r_1 must remain within the body of its owning class, so that class `A` must be moved with it. However, this constraint is unrelated to access control and therefore out of scope. We will return to this issue in Section 3.2.

```
package a; class A {}
package b; class B extends a.A { protected /*d1*/ void m() {..} }
package b; class C { void n() { (new B()).m(); } }
```

pulling up d_1 is correctly prevented by Acc-1 (but nevertheless performed without warnings by ECLIPSE, NETBEANS, and IDEA). However, Acc-1 is insufficient to maintain inheritance under refactoring, as the following example shows (note how `i` can be accessed from `B` even though `i` is protected and `B` is in a different package):

```
package a;
public class A {
  protected /*d1*/ int i;
  void n() { /*r1*/ (new b.B()).i = 1; }
}

package b;
public class B extends a.A {}
```

Here, reducing the declared accessibility of d_1 produces an error, even though $\alpha(\lambda(r_1), \lambda(d_1)) = \mathit{private}$. The reason for this is that after the reduction, `B`, the type through which `i` is accessed, no longer inherits `i`. The reduction of accessibility and the concomitant loss of inheritance are prevented by the constraint rule

$$\beta(r) = d \wedge \rho(r) \neq \lambda(d) \quad \Rightarrow \quad \langle d \rangle \geq \alpha(\rho(r), \lambda(d)) \qquad \textbf{(Inh-1)}$$

which, in the above example, requires at least *protected* for $\langle d \rangle$. As above, Inh-1 does not replace Acc-1 in case of accessing inherited members — it adds to it, effectively requiring that $\langle d \rangle$ is greater than the maximum of $\alpha(\lambda(r), \lambda(d))$ and $\alpha(\rho(r), \lambda(d))$.

However, there is another problem with inheritance, namely that access of a static field can become ambiguous if it is inherited both from a superclass and from an interface [6, §8.3.3.3]. For instance, in

```
class A { private /*d1*/ static int i = 1; }
interface I { /*d2*/ static int i = 2; }
class B extends A implements I {void m() { int j = /*r1*/ i; } }
```

in which $\beta(r_1) = d_2$, the accessibility of d_1 must not be increased**.** While the compiler detects and denies such ambiguous access, a refactoring changing the accessibility of the field in the superclass so that it is inherited by the subclass (where it was not prior to the refactoring) must foresee this problem and refuse its application. This is achieved by the constraint rule

$$\{d, d'\} \subseteq F \cap S \wedge \iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in \text{superclasses}(\rho(r)) \quad \Rightarrow \\ \langle d' \rangle < \alpha(\lambda(r), \lambda(d')) \qquad \textbf{(Inh-2)}$$

in which $\iota(d)$ refers to the unqualified identifier (simple name) of d and superclasses(.) has the obvious meaning (analogous to subclasses(.) in Acc-2). Note that for qualified references r to d, $\rho(r)$ corresponds to an interface, so that Inh-2 is not applicable (because superclasses($\rho(r)$) is undefined). Also note that $\langle d' \rangle$ depends on $\lambda(r)$, not $\rho(r)$, since access, not inheritance, may become ambiguous.

There is a variant of the above example in which d_1 does not exist in class `A` prior to the refactoring, for instance because it is yet to be pulled up from a subclass. To prevent such a refactoring (which would affect the binding of r_1), Inh-2 must be applied to a declared entity d' (d_1 in the above example) that is not yet there (or, rather, that has as yet another location), so that r cannot yet bind to it. We call such constraint rules, which anticipate a refactoring, *foresight rules*. They can only be applied when

the intended refactoring is known.

Subtyping A rather straightforward constraint rule expresses that in JAVA, the accessibility of an overriding or hiding method must not decrease ([6, §8.4.8.3]). This is expressed by the constraint rule

$$\{d, d'\} \subseteq M \wedge (\text{overrides}(d', d) \vee \text{hides}(d', d)) \quad \Rightarrow \quad \langle d' \rangle \geq \langle d \rangle \qquad \textbf{(Sub-1)}$$

in which overrides(.,.) and hides(.,.) have the obvious meanings. Note that the subtyping rule does not apply to fields in JAVA; however, as we will see below, hiding (including that of fields) gives rise to another constraint rule. Also note that, as for Inh-2 above, there is a foresight application of this rule, namely when the method *d* is pulled up from a sibling class.

A rather subtle implication of subtyping in JAVA is that a method inherited by a class that implements an interface requiring that method must remain publicly accessible. This is expressed by the constraint rule

$$\{d, d'\} \subseteq M \wedge \text{subsignature}(d', d) \wedge \{c, c'\} \subseteq C \wedge i \in I \wedge \lambda(d) = i \wedge \lambda(d') = c' \\ \wedge \text{implements}(c, i) \wedge \text{inherits}(c, d', c') \quad \Rightarrow \quad \langle d' \rangle = \text{public} \qquad \textbf{(Sub-2)}$$

in which subsignature(.,.) is defined as in [6, §8.4.2] and implements(.,.) as well as inherits(.,.,.) have their obvious meanings.

Dynamic binding Since constraints work in both directions, the above subtyping constraint rule Sub-1 equally states that the access modifier of an overridden method must always be less than or equal to that of the overriding method. Thus, if the access modifier for an overriding method should be decreased for any reason, the access modifier of the overridden method may also have to decrease.

There are however bounds to this decrease, set by JAVA's rules for dynamic binding. For example, given the JAVA code

```
class A {
  /*d1*/ void m() {..}
  void n() {/*r1*/ m();}
}
class B extends A {
  /*d2*/ void m() {..}
}
```

changing accessibility of d_1 to *private* is syntactically correct, but changes the meaning of the program, since the call of m() in n() is no longer dispatched to the implementation of m() in B, if n() is invoked on an instance of B. Therefore, we add a constraint rule

$$\text{overrides}(d', d) \quad \Rightarrow \quad \langle d \rangle \geq \alpha(\lambda(d'), \lambda(d)) \qquad \textbf{(Dyn-1)}$$

This models the requirement that for a method to be overridden, it must be accessible from the overriding subclass [6, §8.4.8.1]. Note that whether the loss of dynamic binding actually leads to a change of meaning of the program depends on the dynamic types of the receiver objects, and thus on conditions that cannot generally be decided statically. Therefore, Dyn-1 is a conservative rule that prohibits illegal refactorings, but may also prevent legal ones.

Accidentally losing overriding and dynamic binding has a converse problem, namely accidentally introducing it: if in the program

```
class A {
  private /*d1*/void m() {..}
  void n() {/*r1*/m();}
}
class B extends A {
  /*d2*/void m() {..}
}
```

accessibility of d_1 is increased to *package*, the meaning of the program changes for invocations of `n()` on instances of `B`. This is prevented by the constraint rule

$$\lambda(d) \in \text{superclasses}(\lambda(d')) \wedge \text{subsignature}(d', d) \wedge \neg\text{overrides}(d', d) \quad \Rightarrow$$
$$\langle d \rangle < \alpha(\lambda(d'), \lambda(d)) \qquad \textbf{(Dyn-2)}$$

Note that Dyn-1 and Dyn-2 are not only useful for preventing a change of access modifiers that changes the status of dynamic binding — they are also capable of correcting access modifiers when moving subclasses to other packages, so as to maintain (absence of) overriding. For instance, Dyn-1 requires increasing the access modifier of d_1 in

```
public class A {
  /*d1*/void m() {..}
  void n() {
    A a = new B();
    /*r1*/a.m();
  }
}
public class B extends A {
  /*d2*/void m() {..}
}
```

to *protected* when class `B` is moved to another package, which otherwise would prevent execution of d_2 (example adapted from [14]). Note that the concomitant required increase of the accessibility of d_2 is mandated by Sub-1, requiring that $\langle d_2 \rangle \geq \langle d_1 \rangle$.

Further note that because Dyn-1 and Dyn-2 have mutually exclusive antecedents, they can never introduce a direct (i.e., not involving other declared entities or references) contradiction. This is different, however, for Sub-1 and Dyn-2: since their antecedents can both be fulfilled for the same pair (d, d'), one might be concerned about unforeseen interactions. However, due to the declarative nature of constraints, this is not necessary: if all rules are correct, the result of their combined application is also correct (even if the resulting constraints are unsolvable; see below for an example of this). Constraints are inherently modular.

Overloading In addition to overriding, JAVA allows overloading, which poses its own problems. For example, in the JAVA program

```
class A {
  /*d1*/void m(Object o) {..}
}
class B extends A {
  /*d2*/void m(String s) {..}
}
class C {
  void n() { /*r1*/(new A()).m("abc"); }
}
```

where all classes reside in the same package, $\beta(r_1) = d_1$. However, when d_2 is pulled up from B to A, the binding of r_1 changes to d_2, changing the meaning of the program.

The problem here is similar to that of inheriting two static fields (Inh-2) in that neither reference r nor declared entity d of an existing binding $\beta(r) = d$ changes location or accessibility — instead, a new declared entity d' becomes accessible, affecting the binding of r. Therefore, as with Inh-2 we have to create a constraint limiting the accessibility of the new declared entity d' (or a declared entity in a new location) so that it remains inaccessible for references that would otherwise be re-bound to d'. This is done by the constraint rule

$$\{d, d'\} \subseteq M \wedge \text{overloads}(d', d) \wedge \beta(r) = d \wedge \lambda(d') \in \text{superclasses}(\rho(r)) \cup \{\rho(r)\} \quad \Rightarrow \\ \langle d' \rangle < \alpha(\lambda(r), \lambda(d')) \qquad \textbf{(Ovr)}$$

in which overloads(d', d) is defined as in [6, §8.4.9]. As for Inh-2, a constraint generated from Ovr constrains accessibility of a declared entity in a constraint set in which variable values have been updated to reflect the refactoring (in the above example, $\lambda(d_2)$ has changed to a new location). The constraint may be invalid before the refactoring in the sense that it does not adequately reflect the program as is (in the above example, there is no reason to restrict, on the basis of r_1, accessibility of d_2 where it *is* located). Because its application must foresee the refactoring to be performed, Ovr is a foresight rule that, like Inh-2 and Hid, can only be applied to a program when the planned refactoring is known.[8]

The overloading constraint rule Ovr has an interesting consequence: it can require access modifiers to be less than *private*, which basically means that the so modified entity must not be there. While this may seem paradoxical, it makes perfect sense in certain situations: for instance, if $\lambda(r_1)$ in the above example were class A, Ovr would produce $\langle d_2 \rangle < private$, meaning that m(String) must not be declared in A (which is the only correct solution to the problem). In order for all constraints generated by our rules to be satisfiable, we introduce a new value to our set of access modifiers, *A*, which is smaller than *private*. We call this access modifier *absent*.[9] Note that a constraint requiring an existing declared entity to be absent can only be generated by foresight rules (because otherwise the program from which it were created, having an entity it must not have, would be incorrect), and that no constraint variable can have the initial value *absent*.

Hiding The overloading rule Ovr has another interesting application: if we extended the antecedent to cover overriding methods, Ovr could prevent the pulling up of d_2 in

```
class A {
  /*d1*/void m() {..}
}
class B extends A {
  void n() { /*r1*/m(); }
}
```

[8] Note that the converse problem, namely that binding of r to d' is redirected to d because d' became inaccessible, is prevented by Acc-1.

[9] Satisfiability with abnormal values like *absent* is different from lack of satisfiability, since it provides a diagnosis of the problem and points to a possible solution. Cf. Section 6.1 for a discussion.

```
class C extends B {
  /*d2*/ void m() {..}
}
```

which would lead to a change of binding of r_1. The pulling up would be prevented by Ovr because its application would produce the constraint $\langle d_2 \rangle < \alpha(\lambda(r_1), \lambda(d_2))$, which is at conflict with the constraints generated by Sub-1, $\langle d_1 \rangle \leq \langle d_2 \rangle$, and Acc-1, $\langle d_1 \rangle \geq \alpha(\lambda(r_1), \lambda(d_1))$ $(= \alpha(\lambda(r_1), \lambda(d_2)))$, so that the refactoring would lead to an unsolvable constraint set (meaning that pulling up d_2 is not allowed).

Rather than extending Ovr as suggested above, we introduce a separate constraint rule that also covers static methods and fields, whose introduction in a type can likewise lead to a change of binding (called *hiding* in [6] or *hiding through inheritance* in [13]). However, other than with overloading, with hiding there will never be solutions consisting of reducing accessibility of the hiding declared entity to a level above *absent*. Therefore, the new constraint rule reads

$$\iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in (\text{superclasses}(\rho(r)) \cup \{\rho(r)\}) \setminus \text{superclasses}(\lambda(d)) \Rightarrow \langle d' \rangle = \mathit{absent} \quad \textbf{(Hid)}$$

where $\iota(d) = \iota(d')$ means that d and d' have the same name or are override-equivalent [6, §8.4.2]. Again, that a declared entity that hides must be absent may seem paradoxical, but just as with inheritance (Inh-2) and overloading (Ovr), Hid is not applied to a program as is, but rather to the changes introduced by the refactoring were it performed (a kind of internal preview). It is thus a foresight rule, here one preventing that a certain declared entity is introduced, or moved, to a certain location.

Note that so-called *shadowing* and *obscuring* [6] (called *hiding by nesting* in [13]) cannot be prevented by adjusting access modifiers and are therefore out of scope for this paper.

Miscellaneous A number of constraints follow directly from the JAVA language specification (JLS) [6] and are easily formalized:

- The accessibility of an array type equals the accessibility of its element type [6, §6.6.1].
- The accessibility of all fields declared in the same field declaration must be equal [6, §8.3].
- All main methods must be publicly accessible [6, §12.1.4].
- Only that top level type of a compilation unit whose name equals the name of the compilation unit may be declared public [6, §7.6].[10]
- A singly imported type and imported static members must be accessible by the importing compilation unit [6, §7.5].

Not so easily formalized (and omitted here for spatial reasons) is the rule that for multiple on-demand imports [6, §7.5], if a simple name in the importing compilation unit refers to a declared entity imported by one of the imports, the same entity must not be accessible through any of the other on-demand imports [6, §6.5].

For open programs (libraries, frameworks, etc.) it is necessary to keep other entry points than the main methods accessible. We therefore interpret certain annotations as constraints keeping the accessibility of the annotated entity constant; the @API annota-

[10] Note that in our formalization, compilation units have not been included as locations.

tion introduced in [1], as well as the `@Test` annotation of JUNIT, are examples of this.

Last but not least (and as announced in Footnote 4), the set of admissible access modifiers for a declared entity depends on its kind and where it is declared, which is modelled by a corresponding constraint rule. Note that the allowable modifiers of *d* may change should the location of *d* change, for instance when *d* is a method pulled up to an interface.

3.2 Solving accessibility constraints

Solutions to finite constraint sets over variables with finite domains are trivially found by generating all possible variable assignments and by testing for each assignment whether it solves the constraint set. Clearly, the computational complexity of such a procedure is exponential in the number of variables, and therefore rarely acceptable. However, while general constraint satisfaction problems are known to be NP-complete, in practice, highly efficient algorithms that can solve finite domain constraint satisfaction problems such as ours with thousands of variables in acceptable time are available off the shelf (see, e.g., [5]), so that we will not go into details here. With one notable exception.

Since our constraint rules only model one aspect of JAVA, namely its access control, the constraints generated from these rules cannot be expected to prevent changes to programs violating syntactic or semantic rules unrelated to accessibility (examples of this are given in Footnote 7 and in Section 6.3). In particular, generated constraint sets may have solutions involving the changed location of elements that translate to incorrect programs, even if no access constraint is violated. Therefore, we restrict constraint solving to computing new values for the variables representing the declared access modifiers of entites, $\langle . \rangle$, and keep the variables representing locations of declared entities and references, $\lambda(.)$, constant (unless of course the change of location is the purpose of a refactoring). Since the sets of possible locations for references and declared entities are usually large, this reduces the complexity of our constraint satisfaction problems considerably.

4 Refactoring with accessibility constraints

Traditionally, the specification of a refactoring consists of a set of preconditions and an algorithmic part that describes its "mechanics" [4].[11] The preconditions are checked before the refactoring is performed; their purpose is to exclude applications of the refactoring to constellations in which the refactoring cannot work.

For constraint-based approaches to refactoring, precondition checking and mechanics rely on the same characterization of the problem: precondition checking amounts to finding out whether a generated constraint set with the intended changes applied is solvable, and performing the mechanics amounts to writing a solution of the constraint set (i.e., the found variable values) back to the program. If checking solvability

[11] Being an algorithm, the complete specification of a refactoring would also involve a set of postconditions. However, postconditions of refactorings are rarely found in the literature.

and finding a solution are considered one, refactoring with constraints consists of four steps:

1. the generation of constraints and initial variable values from the program to be refactored (resulting in a solved constraint set);
2. a change of variable values and the addition of foresight constraints reflecting the planned refactoring (possibly resulting in an unsolved constraint set);
3. the solution of the constraint set under the side conditions of the refactoring (including which constraint variables are fixed and which can be changed as part of the solution); and
4. the writing back of the found solution, if any.

The refactoring may involve user interaction, namely answering questions as to whether certain changes should be allowed (such as the change of access modifiers due to ripple effects). Since the constraints required for each particular refactoring, which variables of these constraints are actually changeable, and the possible user interaction depend on the concrete refactoring, we clarify these issues separately for each refactoring.

4.1 The CHANGE ACCESSIBILITY refactoring

The most primitive refactoring relating to accessibility is changing the access modifier of a declared entity. It is a refactoring because, as noted in the introduction, the change represents a change of design and because it requires a careful prior analysis (it can change the meaning of the program, which a refactoring must not do).

The constraints to be generated for this refactoring are those involving the entity d whose declared accessibility $\langle d \rangle$ is to be changed, and recursively all those that are directly or indirectly (through shared constraint variables) related to it. If the user chooses that no other declared entities d' may be touched in the course of the refactoring, the set of constraints needing to be generated is reduced to the ones in which d is directly involved; the declared access modifiers $\langle d' \rangle$ of the d' participating in these constraints are then marked as constant.

The computation of the solution is initiated by assigning the constraint variable $\langle d \rangle$ representing the declared accessibility of the entity d to be refactored the value corresponding to the target accessibility. If the new value leaves the constraint set solved, the changed value can be written back and the refactoring is performed. If it is unsolved, a new solution must be computed. To express that the solution should involve as few and as small changes as possible (a side condition of the refactoring), the number of changes must be counted and the constraint solver instructed to find a solution that minimizes this count. If no solution exists, the refactoring must be refused; otherwise, the solution is written back and the refactoring performed.

Regarding the foresight rules preventing a change of binding (Inh-2 and Ovr; Hid is irrelevant here, because it can only prevent changes of location), only those constraints need be generated that constrain declared entities d whose accessibility may change during the course of the refactoring. For Inh-2, this amounts to checking all superclasses of $\rho(r)$ for all r with $\beta(r) = d$, for the presence of a static field with the same name as d. If present, a corresponding constraint is added. For Ovr, the check is analogous, but limited to overloaded methods. For instance, if in the program

```
package a;
public abstract class A {
  protected /*d1*/abstract void m(String s);
}

package a;
public class B extends A {
  public /*d2*/void m(Object o) {..}
  protected /*d3*/void m(String s) {..}
}

package b;
public class C {
  void n() { /*r1*/(new a.B()).m("abc"); }
}
```

accessibility of d_1 is to be increased to *public*, application of Ovr adds the constraint $\langle d_3 \rangle < \alpha(\lambda(r_1), \lambda(d_3))$ (because Sub-1 inserted $\langle d_3 \rangle \geq \langle d_1 \rangle$, implying a change of $\langle d_3 \rangle$). Since $\alpha(\lambda(r_1), \lambda(d_3)) = public$, the refactoring is prevented.

A special case of changing accessibility is the HIDE METHOD refactoring, which suggests making "each method as private as you can" [4]. In previous work of ours, we have implemented a refactoring that attempts to hide all methods of a program in one step [1]. However, due to the imperative character of the implementation (cf. Section 2.2), it did not consider ripple effects (i.e., one reduction that required another reduction as a prior step), so that it had to be repeated after each application to see whether any new reductions had become possible. By contrast, the constraint approach we are presenting here addresses this chaining through constraint propagation, so that accessibility of all declared entities can be reduced to their smallest possible levels in a single refactoring step.

4.2 The MOVE TYPE/MEMBER and PULL UP/PUSH DOWN MEMBER refactorings

As far as accessibility is concerned, there is no difference between moving and pulling up or pushing down members (a distinction that is made in [4] and also in many refactoring tools): the constraint rules to be applied are precisely the same. The difference lies in which other elements must be moved as well, but this is independent of access modification and hence outside the scope of our work (cf. Sections 3.2 and 6.3). Therefore, we do not distinguish between these refactorings here.

Moving one or more declared entities d means that the constraint variables representing their locations, $\lambda(d)$, are assigned new values. If the declared entities contain references r, their locations $\lambda(r)$ change as well. All constraints directly or indirectly involving the changed $\lambda(d)$ or $\lambda(r)$ must be generated. If the user selects that no access modifiers may be changed as part of the refactoring, the set of constraints to be generated is restricted to the ones directly involving the moved locations and the values of all variables $\langle d \rangle$ are considered constant. All this is more or less analogous to the CHANGE ACCESSIBILITY refactoring (cf. above).

The situation is significantly different, however, for the foresight rules: here, Inh-2, Sub-1, Ovr, and Hid must be applied to the *moved* program elements, referring to their updated locations. This implies that search for overloaded or override-equivalent methods and for fields of the same name must be commenced in the target, rather than the original, location. For instance, if the intended refactoring for the program

```
package a;
public class A {
  public /*d1*/void m(Object o) {..}
  void n() { /*r1*/(new b.B()).m("abc"); }
}

package b;
public class B extends A {}

package a;
public class C extends B {
  public /*d2*/void m(String s) {..}
  void n() { /*r2*/(new C()).m("abc"); }

}
```

is to pull up d_2 to class B, application of Ovr to r_1, d_1, and d_2 in its new location, class B, produces the constraint $\langle d_2 \rangle < \alpha(\lambda(r_1), \lambda(d_2))$ (= *public*; note that this constraint is not justified for the program before the refactoring). Since application of Acc-1 produced $\langle d_2 \rangle \geq \alpha(\lambda(r_2), \lambda(d_2))$ (= *protected*), the pulling up of d_2 is possible, but only if $\langle d_2 \rangle$ is reduced to *protected*. Since we restricted the moving of program elements to the ones the user required to be moved (cf. Section 3.2), for all others the foresight rules must be applied as if the refactoring were CHANGE ACCESSIBILITY (because this is all that can happen). In the above example, there are no other foresight rules to be applied.

4.3 Renaming declared entities and changing method signatures

A number of standard refactorings have the potential to change bindings. Perhaps the most prominent is the RENAME refactoring, which has to deal with issues such as hiding, shadowing, and obscuring [6, 18]. In certain cases, a change of accessibility can prevent such changes of binding, but these cases are rather rare. Also, the choice of names should not have an impact on accessibility and thus modularity, so that we do not pursue this further here.

Somewhat related is the problem of changed bindings due to a change of method signatures, either due to user request or as a side effect of refactorings such as GENERALIZE DECLARED TYPE or USE SUPERTYPE WHERE POSSIBLE [21]. In languages with single dispatch, the change of binding is limited to overloaded methods, and therefore can be dealt with using our constraint rules (in particular Ovr). However, as will be discussed in Section 6.2, other means of preventing or solving such problems may be more adequate.

4.4 Enabling other refactorings

Although not themselves concerned with changing access modifiers, some refactorings have preconditions requiring a certain level of accessibility of involved declared entities. For instance, the REPLACE INHERITANCE WITH DELEGATION [10] refactoring requires that the inheriting class or its subclasses do not need access to protected members inherited before the refactoring (because these are no longer accessible after inheritance has been replaced by delegation). A corresponding case study showed that

Table 1. Space and time requirements of the approach as currently implemented (see text).

	SPACE			TIME			
	No. of			*Avg. Time*[$] *in msec to*			*Avg. No. of*
Project	λ(.)	⟨.⟩	*Constraints*	*Build*	*Check*[*]	*Solve*[*]	*Steps to Solve*[*]
JUNIT 3.8.1	2553	1332	4949	10593	30	599	5293360
JESTER 1.37b	1475	761	2293	1127	9	81	6837
JHOTDRAW 6.01b	9594	4995	26816	21246	199	6452	25582800
APACHE.IO 1.4	4315	2181	12877	17843	129	2486	57052

[$] on a contemporary Wintel machine with 2GHz clock speed and 1GB of main memory for the JVM
[*] averaged over the refactorings performed to obtain the data of Table 2

in more than 15% of all inheriting classes, this precondition was violated [10]. Increasing accessibility of the (formerly) inherited member to *public* would satisfy the precondition; however, this presupposes that such a change does not change the meaning of the program. This can easily be checked by our constraints.

5 Implementation

As a proof of concept, we have implemented our constraint-based model of accessibility in JAVA as a plug-in to ECLIPSE's JDT, and tested and evaluated it by using it as the basis of several systematic refactorings of a set of sample programs.

5.1 Constraint generation

Section 4 described in abstract terms how the constraints required for a specific refactoring are determined. Basically, an implementation would have to start with an initial set of variables whose change of value models the intended refactoring, generate all constraints from the program that constrain these variables, add their other (changeable) variables to the variable set, and so forth until no more constraints can be added. For instance, if the declared accessibility $\langle d \rangle$ of an entity d is to be changed, the program must be scanned for matches of the preconditions of all constraint rules containing d. For Acc-1 with precondition $\beta(r) = d$ this means that all references r binding to d must be found, for Inh-1 that additionally the static type of the receiver must be looked up, and so forth. As it turns out, the required searches and lookups can be quite expensive, especially if the AST does not maintain inverted indices pointing from declared entities to their references (cf. the discussion in Section 2.2). Since the space and time requirements for building and keeping such indices can be substantial (see [1] for some measurements), and since the JDT's search functions also rely on scanning the AST (so that successive searches for references to different declared entities are rather expensive), we decided to generate all constraints in a single sweep of the AST, regardless of whether they are actually needed by the concrete refactoring problem. As can be seen from Table 1, this poses some non-negligible spatial and temporal limits on our implementation.

5.2 Auxiliary functions

Except for α, the auxiliary functions and predicates occurring in the antecedents of our constraint rules (namely β, λ, ρ, ι, subclasses, superclasses, implements, inherits, hides, overrides, overloads, and subsignature) are implemented using corresponding API methods of the JDT. The function α computing the required accessibility level for a reference to a declared entity is unpleasant to specify (as was its extraction from the JLS [6]); because it is of no theoretical interest, we do not present it here.

5.3 Constraint solution

For constraint set solution, we adopted Naoyuki Tamura's class library for constraint programming in JAVA, called CREAM [20]. CREAM offers various implementations of efficient solvers for finite domain (especially integer) constraint satisfaction problems, of which our accessibility constraint sets (with the finite and totally ordered A as their domain) are a special case. As can be seen from Table 1, CREAM is capable of computing a new solution for a constraint set invalidated by the change of variable values and the addition of foresight constraints modelling an intended refactoring in acceptable time. However, better performance can be expected from creating fewer constraints (cf. Section 5.1), and from devising problem-specific constraint solvers.

5.4 Testing

In the absence of a formal proof of the completeness and correctness of our constraint rule set, we tested it thoroughly, exploiting the invariants mentioned at the end of the introduction to Section 3. In particular:

1. We generated all constraints and initial variable values from existing programs and checked whether the resulting constraint sets were solved given the initial assignments. This gave us an idea of the correctness of our constraint rules.
2. We computed all solutions for constraint sets generated from programs covered by test suites and wrote back the solutions to the code, checking whether the programs still compiled and their test suites still passed. This gave us an idea of the completeness of our constraint rule set. Note that, because behaviour-preserving change of location is not only constrained by accessibility (cf. Sections 3.2 and 6.3), we only computed new values for the variables $\langle . \rangle$ representing the declared access modifiers.[12]
3. We automatically performed refactorings enhanced with accessibility constraints for precondition checking and for computing the necessary mechanics on several programs covered by accompanying test suites, and checked whether the refactorings left the meaning of the programs (as specified in the test cases) unchanged.

[12] Due to the exponentially growing number of possible solutions, we had to limit testing to small programs (mostly variants of the programs used as examples in this paper, but also subsets of JUNIT and other small programs). We complemented these tests by tests on much larger programs, in which we changed only one access modifier at a time.

All three approaches contributed to identifying and shaping our constraint rules, whose original extraction from the JLS [6] turned out to be difficult and error-prone.

For the automated application of refactorings to sample programs, we used our REFACTORING TOOL TESTER (RTT) program. RTT is an ECLIPSE plugin that automatically applies a given ECLIPSE refactoring tool to all those elements of a test program for which it is intended, and checks whether the program still compiles after the refactoring and whether its unit tests still pass (approximating behaviour preservation). In its purpose, RTT competes with ASTGEN [2], but its design is different in that it uses existing, rather than specially generated, programs. Using the RTT on a large set of test programs increases the likelihood of covering rare cases (the designers of the generators for ASTGEN may not even have conceived of). However, since this approach primarily tests the refactorings and only indirectly the constraints used to compute preconditions and necessary changes, and because these refactorings have bugs unrelated to accessibility, the results of this automated testing require a careful interpretation. We defer this to the next subsection.

5.5 Evaluation

Although our examples of Section 2.1 should have provided sufficient evidence for the usefulness of a formal capture of accessibility and its integration into refactoring tools, we have also conducted some experiments using real programs, giving us an impression of how often a user of these tools will actually benefit. For this, we have adapted our above described RTT to apply two ECLIPSE refactoring tools, MOVE CLASS and PULL UP METHOD, in three variants to a set of sample programs: variant 1 (*pure*) applies the refactorings as they are currently deployed with [3] (including their built-in precondition checking); variant 2 (*prec*) enhances them with our constraint-based precondition checking allowing no changes of access modifiers, and variant 3 (*mech*) enhances them with precondition checking and constraint-solution based mechanics adjusting access modifiers so as to make the refactoring possible. Thus, we get for each potential application (*appl*) of a refactoring six outcomes, namely for each variant one pair stating whether it passed the preconditions (*p*) and whether it was successful (*s*). The counts of these outcomes for the sample programs are summarized in Table 2.

Due to the nature of the problem, we can expect the number of precondition passes, *p*, to decrease from *pure* to *prec*, and the number of successes, *s*, to stay the same (if it decreased, our preconditions would likely be too strong). Thus, the relative number of successful applications should increase. When moving from *prec* to *mech*, *p* should increase, as should *s*. However, because the refactorings can fail for other reasons (see below), the relative number of successful applications can change in either direction.

For MOVE CLASS, the results from Table 2 confirm our expectations: the passing of preconditions drops by 15% on average when moving from *pure* to *prec*[13], and increases by 17% when moving from *prec* to *mech*. This means that if the refactoring is allowed to change access modifiers, preconditions predict that it can be applied in 99% of all cases, compared to 85% if no changes are allowed. The success rate, which

[13] The inhibiting constraint rules were Acc-1 (52×), Dyn-1 (7×), and Inh-1 (4×).

Table 2. Number of passed (*p*) and successful (*s*) refactorings as applied to several test projects (see text).

	MOVE CLASS							PULL UP METHOD						
	appl	*pure*		*prec*		*mech*		*appl*	*pure*		*prec*		*mech*	
Project		*p*	*s*	*p*	*s*	*p*	*s*		*p*	*s*	*p*	*s*	*p*	*s*
JUNIT 3.8.1	38	38	23	23	23	36	36	148	20	20	14	14	20	20
JESTER 1.37b	31	31	26	26	26	31	31	5	3	3	3	3	3	3
JHOTDRAW 6.01b	235	235	213	208	208	235	235	1167	199	187	147	139	168	160
APACHE.IO 1.4	73	73	64	64	64	73	73	102	14	14	14	14	14	14
total	377	377	326	321	321	375	375	1422	236	224	178	170	205	197

is only 86% for *pure*, increases to 100% for *prec* and *mech*. The decrease of successful applications from *pure* to *prec* by 2% in the case of JHOTDRAW turned out to be due to measurement error: the five surplus applications of *pure* changed the meaning of the program (an were thus in fact unsuccessful), but this was neither caught by the compiler (in the form of compile-time errors) nor by the test cases. The two illegal applications of *pure* that could not be legalized by *mech* (both from JUNIT) were due to unsatisfiable constraints introduced by application of Acc-1 and Dyn-2.

The picture is rather different for PULL UP METHOD: while applicability also decreases from *pure* to *prec* (by 24% on average[14]), the loss of applicability is not reverted by *mech*: of the 58 applications inhibited by *prec*, only 27 could be legalized by adapting access modifiers. The remainder was prevented by unsatisfiable constraints of the above kind, and also by the 16 constraints introduced by Hid (cf. Footnote 14), which are generally unsatisfiable for referenced entities (cf. Section 3.1). The high success rate of *pure* (95%) is explained by the fact that the original refactoring tool changes the access modifier of the pulled up method if this is deemed necessary (which avoids many compile-time errors), and that introduced binding errors are not caught by tests (recall that Dyn-1 and Dyn-2 may be too strict so that their violation may not even present an error). The eight unsuccessful applications for *prec* and *mech* are caused by errors introduced by the pure refactoring tool that are unrelated to accessibility (and thus can neither be prevented by *prec* nor fixed by *mech*), such as disregarding the changed type of this (cf. Section 6.3) and the incompatibility of exceptions thrown by the pulled up method and an override-equivalent method in a sibling class.

6 Discussion

6.1 The value of *absent*

The existence of *absent* as an access modifier allows the elegant formulation of certain preconditions of refactorings as *solvable* constraints (cf. Footnote 9). For in-

[14] with Ovr (365×), Acc-1 (125×), Sub-1 (18×) and Hid (16×) inhibiting the application

stance, if a foresight constraint requires that a certain entity's declared accessibility must be less than *private*, and if no other constraint requires that it must be at least *private* (because the entity is never referenced), the constraint solver will assign it the value *absent*, suggesting that the element should be (and can be!) deleted. Without *absent*, the constraint set would be unsolvable and the possible solution, the removal of a declared entity, would remain unconsidered.

Beyond this, *absent* also has its own value. Analogous to JAMIT [8], our constraint rules are capable of detecting dead code, simply by searching for solutions of a program's constraint set that assigns *absent* to the access modifiers of declared entities. However, with the access rules as is, dead code can remain undetected due to circular referencing.

Declared entities that are sustained by circular references not fed by a reference into the circle can be reduced to *absent* by modifying the accessibility rule Acc-1 such that it allows *absent* for a $\langle d \rangle$ even though there is an r such that $\beta(r) = d$, if r resides in the location of a declared entity that is itself modified with *absent* (expressed by absent(r)):

$$\beta(r) = d \quad \Rightarrow \quad \langle d \rangle \geq \alpha(\lambda(r), \lambda(d)) \vee \langle d \rangle = \mathit{absent} \text{ if absent}(r) \qquad \textbf{(Acc-1')}$$

However, since all constraints are generated by a rather simple static analysis of the program, our dead code removal will always be inferior to that achieved by more sophisticated tools, so that we do not pursue this further here.

6.2 Too strong preconditions

One might argue that the requirement expressed by some of our constraint rules, that declared entities must be made inaccessible or even eliminated to allow certain refactorings, is unnecessarily strong. Indeed, the reference to an overloaded method can be forced to bind to a certain implementation by inserting upcasts to the formal parameter types of that particular implementation, and the reference to a hidden entity can be maintained by inserting qualified names (as described in [18] for the RENAME refactoring; cf. Section 2.2). However, this would require a change of the reference rather than a change of accessibility, and is therefore a different story (one in which references themselves are modelled as variables).

6.3 Boundaries of accessibility constraints

Controlling access modifiers does not solve all refactoring problems related to accessing members. For instance, in the left program of

```
class A {}

class B extends A {
  /*d1*/ void m() {/*r1*/ n();}
  /*d2*/ void n() {}
}
```

```
class A { B b; }

class B extends A {
  /*d1*/ void m() {/*r1*/ b.n();}
  /*d2*/ void n() {}
}
```

with classes `A` and `B` residing in the same package, pulling up `m()` does not violate the access rules of JAVA (`B.n()` is accessible from `A`), but nevertheless results in a seman-

tic error — the problem here is that the implicit receiver of calling n(), this, is of type B before the refactoring and of type A after. Were n() called on a variable of type B as on the right, the program would still work after the refactoring. While a constraint-based solution to this problem is likely possible, it is independent of access modification and therefore out of scope for this paper.

6.4 Formal comparison of languages

Our capture of access control as a set of constraint rules allows the compact comparisons of programming languages. Table 3 provides such a comparison of JAVA, C#, and EIFFEL. The inclusion of EIFFEL may seem surprising, since EIFFEL does not have access modifiers, but uses selective export of features (to the listed classes and their subclasses, to all, or to no classes) instead [12]. However, it nevertheless fits nicely into our framework: in EIFFEL, the domain of the accessibility variables $\langle . \rangle$, A, is $\wp(C)$, the powerset of the set of classes (see Table 3).

Table 3 also reveals that C#, although rather similar to JAVA, avoids certain of its problems. For instance, violation of Dyn-1 resulting in a change of behaviour in JAVA leads to a semantic error reported by the compiler, and violation of Dyn-2 is impossible. Similarly, violation of Hid issues a warning suggesting that the new modifier be used. A constraint rule of C# not found in JAVA requires that the accessibility of a member is at most the accessibility of its declared type. This prevents the breaching of non-accessibility made possible by chained method calls in JAVA, through which an instance of an inaccessible type can be accessed.

7 Conclusion

Refactoring the design of a program typically involves the moving of classes and/or their members. This requires regard of the access control rules specified by the programming language. In JAVA, disregard of these rules cannot only lead to access violations (reported as errors by the compiler), it can also lead to a change of meaning of a program, which a refactoring must always avoid. By capturing the access control of JAVA in the form of constraint rules, we have provided a framework for checking the preconditions of refactorings affecting the accessibility of program elements, and for safely adapting declared accessibility as part of the mechanics of a refactoring that would otherwise be impossible. Our framework involves so-called foresight applications of rules that model the changes intended by a refactoring, and an additional access modifier *absent* suggesting the deletion of program elements that are in the way of a refactoring without being used by the program. By conducting systematic experiments, we have shown how our approach can improve applicability and correctness of at least one important refactoring tool; where it falls short, it may be possible that additional constraint rules (unrelated to access modification) can fix the problems.

Table 3: Comparing JAVA, C#, and EIFFEL by contrasting their accessibility constraint rules

	JAVA	C#	EIFFEL
A	set of access modifiers of which $\langle . \rangle$ is an element *public* > *protected* > *package* > *private*	*public* > *protected internal* > {*protected*, *internal*} > *private*	$\wp(C)$
Acc-1	accessing $\beta(r) = d \;\Rightarrow\; \langle d \rangle \geq \alpha(\lambda(r), \lambda(d))$	same as JAVA	$\beta(r) = d \;\Rightarrow\; \lambda(r) \in \langle d \rangle$
Acc-2	$\beta(r) = d \wedge \alpha(\lambda(r), \lambda(d)) = \mathit{protected} \wedge d \notin S \wedge \rho(r)$ $\notin$ subclasses($\lambda(r)$) $\cup$ $\{\lambda(r)\} \;\Rightarrow\; \langle d \rangle = \mathit{public}$	analogous to JAVA, taking *protected internal* into account	not applicable
Inh-1	accessing inherited members $\beta(r) = d \wedge \rho(r) \neq \lambda(d) \;\Rightarrow\; \langle d \rangle \geq \alpha(\rho(r), \lambda(d))$	no additional constraint	no additional constraint
Inh-2	accessing multiply inherited static fields $\{d, d'\} \subseteq F \cap S \wedge \iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in$ superclasses($\rho(r)$) $\Rightarrow$ $\langle d' \rangle < \alpha(\lambda(r), \lambda(d'))$	not applicable (no implementation can be inherited from an interface)	not applicable (features inherited from more than one superclass must be renamed)
Sub-1	subtyping $\{d, d'\} \subseteq M \wedge$ (overrides(d', d) $\vee$ hides(d', d)) $\Rightarrow \langle d' \rangle \geq \langle d \rangle$	$\{d, d'\} \in M \wedge$ (overrides(d', d) $\vee$ hides(d', d)) $\Rightarrow \langle d' \rangle \boxed{=} \langle d \rangle$ (access modifiers of methods cannot be changed in subclasses)	overrides(d', d) $\Rightarrow$ $\langle d' \rangle \supseteq \langle d \rangle$ $\vee$ check for CAT-calls
Dyn-1	losing dynamic binding overrides(d', d) $\Rightarrow \langle d \rangle \geq \alpha(\lambda(d'), \lambda(d))$	same; upon violation, compiler reports an error, since lack of accessibility conflicts with virtual modifier necessary for overriding	not applicable, since all features are inherited and dynamic binding is independent of accessibility
Dyn-2	introducing dynamic binding $\lambda(d) \in$ superclasses($\lambda(d')$) $\wedge$ subsignature(d', d) $\wedge \neg$overrides(d', d) $\Rightarrow \langle d \rangle < \alpha(\lambda(d'), \lambda(d))$	not applicable, since overriding would require adding a virtual modifier	see above
Ovr	overloading $\{d, d'\} \subseteq M \wedge$ overloads(d', d) $\wedge \beta(r) = d \wedge \lambda(d')$ $\in$ superclasses($\rho(r)$)$\cup\{\rho(r)\} \Rightarrow \langle d' \rangle < \alpha(\lambda(r), \lambda(d'))$	same as JAVA	no overloading in EIFFEL
Hid	hiding $\iota(d) = \iota(d') \wedge \beta(r) = d \wedge \lambda(d') \in$ (superclasses($\rho(r)$) $\cup \{\rho(r)\}$)\superclasses($\lambda(d)$) $\Rightarrow \langle d' \rangle = \mathit{absent}$	same as JAVA, but violation issues a warning suggesting use of the new modifier (not for PULL UP)	same as C#, but hiding feature must be introduced using redefine

$\langle d \rangle \leq \langle \text{type}(d) \rangle$ where type(d) is the entity representing the declared type of d

Acknowledgements

We are indebted to Naoyuki Tamura for making available his CREAM library, and for pointing us to its neighbourhood search facility.

References

1. P Bouillon, E Großkinsky, F Steimann "Controlling accessibility in agile projects with the Access Modifier Modifier" in: *Proc. of TOOLS 46* (2008) 41–59.
2. B Daniel, D Dig, K Garcia, D Marinov "Automated testing of refactoring engines" in: *Proc. of ESEC/SIGSOFT FSE* (2007) 185–194.
3. Eclipse Java Development Tools Version 3.4.1 (http://www.eclipse.org).
4. M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).
5. T Frühwirth, S Abdennadher *Essentials of Constraint Programming* (Springer, Berlin 2003).
6. J Gosling, B Joy, G Steele, G Bracha *The Java Language Specification* (http://java.sun.com/docs/books/jls/).
7. C Grothoff, J Palsberg, J Vitek "Encapsulating objects with confined types" *ACM Trans. Program. Lang. Syst.* 29:6 (2007) 32.
8. C Grothoff *Introducing: the Java Access Modifier Inference Tool* (http://grothoff.org/christian/xtc/jamit/).
9. IntelliJ IDEA Version 8 (http://www.jetbrains.com/idea/).
10. H Kegel, F Steimann "Systematically refactoring inheritance to delegation in Java" in: *Proc. of ICSE* (2008) 431–440.
11. L Koved, M Pistoia, A Kershenbaum "Access rights analysis for Java" in: *Proc. of OOPSLA* (2002) 359–372.
12. B Meyer *Object-Oriented Software Construction* 2nd edition (Prentice Hall International, 1997).
13. Microsoft *C# Language Specification v1.2* (http://download.microsoft.com).
14. P Müller, A Poetzsch-Heffter "Kapselung und Methodenbindung: Javas Designprobleme und ihre Korrektur" in: *Java-Informations-Tage* (1998) 1–10.
15. NetBeans Integrated Development Environment Version 6.5 (http://www.netbeans.org).
16. W Opdyke *Refactoring Object-Oriented Frameworks* (Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992).
17. DL Parnas "On the criteria to be used in decomposing systems into modules" *Commun. ACM* 15:12 (1972) 1053–1058.
18. M Schäfer, T Ekman, O de Moor "Sound and extensible renaming for Java" in: *Proc. of OOPSLA* (2008) 277–294.
19. N Schirmer "Analysing the Java package/access concepts in Isabelle/HOL" *Concurrency — Practice and Experience* 16:7 (2004) 689–706.
20. N Tamura *Cream: Class Library for Constraint Programming in Java* (http://bach.istc.kobe-u.ac.jp/cream/).
21. F Tip, A Kiezun, D Bäumer "Refactoring for generalization using type constraints" in: *Proc. of OOPSLA* (2003) 13–26.
22. F Tip "Refactoring using type constraints" in: *Proc. of SAS* (2007) 1–17.