

Type Access Analysis: Towards Informed Interface Design

Friedrich Steimann, Fernuniversität in Hagen, Germany

Philip Mayer, Ludwig-Maximilians-Universität München, Germany

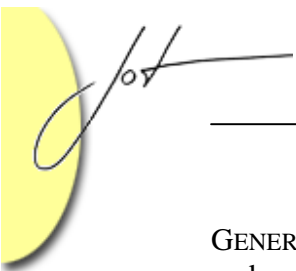
Abstract

Programs designed from scratch often start with just a set of classes. Classes can be instantiated and so deliver the objects that are the carriers of information and function. In languages like JAVA and C++, classes also define types, so that they are sufficient to write a fully functional, type-checked program. Abstract classes and interfaces, which cannot be used for object creation, but which can serve to structure and decouple the code, are then either added later (as a result of refactoring) or never. One impediment to designing and introducing such type abstractions (generalizations) retroactively is that it is unclear how they can be used in a program, or what they should contain in order to be usable. However, this knowledge is, so we argue, completely contained in the program — it only needs to be unveiled. With our Type Access Analyzer (TAA) tool, we collect information useful for the design of type abstractions (abstract classes and interfaces) and their use, and present it to the developer for performing type-related refactorings in an informed manner.

1 INTRODUCTION

JAVA's interface-as-type concept, which seems to be rooted in STRONGTALK [3] and the work of Canning et al. [4], is often thought of as a weak surrogate for multiple inheritance — yet it is really a powerful means of decoupling classes, that is, of avoiding direct dependencies of one class on another. This decoupling is achieved by typing declaration elements (fields, temporaries, formal parameters, and non-void method returns) of a class with interfaces or abstract classes rather than the classes the declaration elements' values are instances of. The more general these types, i.e., the fewer members they declare, the greater is the decoupling achieved, and the greater is the flexibility of the code with respect to future changes and extensions. Possible generalization is limited, however, by the *use of objects* coded in a program, as for a program to be type correct, each type must declare at least what is needed from the objects it comprises.

Various refactoring tools aiding in the creation, maintenance, and use of new interfaces are readily available. Among these are the EXTRACT INTERFACE and PULL UP/PUSH DOWN METHOD refactorings for the manual creation and maintenance of interfaces, and



GENERALIZE DECLARED TYPE and USE SUPERTYPE WHERE POSSIBLE for their automatic and semiautomatic use in variable declarations (all offered, e.g., by ECLIPSE's JDT). Although the automated use of interfaces is a big achievement, their usability presupposes their design for use, which remains a manual, and thus fallible, act.

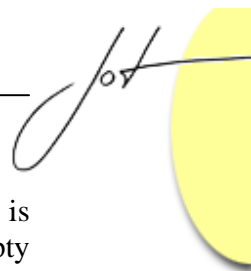
As a first step towards remedying this situation, we created a new refactoring tool — named INFER TYPE [10] — which, based on a declaration element and its use in a program, computes a new, maximally general type for that element. However, application of this refactoring tends to create a distinct type per declaration element (see [18, 19] for a more detailed discussion), which is not what one wants when searching for program-wide abstractions. On the other hand, designing more *globally usable* generalizations requires *global knowledge* that is usually beyond a developer's awareness of a program. In programming practice, this leads to a tedious trial and error process, one in which an interface is created and then continuously refactored until it contains everything that it needs to be usable where it is supposed to be used. We have therefore created a new tool — named TYPE ACCESS ANALYSER (TAA) — that performs an upfront analysis of types and their use in a program, providing data whose systematic exploitation allows informed interface design. We present this tool here for the first time.

In the sections that follow, we lay the theoretical groundwork for the TAA (Section 2), which is based on type inference and the construction of a supertype lattice. Then (in Section 3), we develop a concrete application example that shows how type inference and supertype lattice can be combined to systematically analyse the use of types in a program needing refactoring. As will be seen, the analysis provides the information necessary to change the type hierarchy in various ways, particularly in ways that otherwise require the developer's detailed program knowledge. In Section 4, we discuss the pragmatics of our approach and its limitations. A brief comparison with related work and an outlook to future work conclude our contribution.

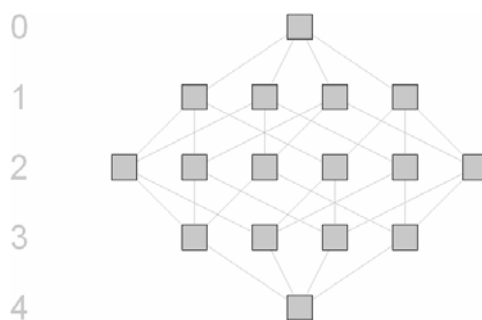
2 SUPERTYPE LATTICE AND INFERRED TYPES

Supertype lattice

Every type defines a set of members. Depending on the programming language, not all members of a type are equally visible — and consequently *accessible* — for other types. We call the set of members of a type that is accessible for other types the *protocol* of that type. Note that in languages like JAVA, what is accessible for other types depends on the location of a type relative to the ones accessing it. For instance, the protocol for the same type includes private members; for types defined in the same package it includes package local members; etc. To simplify matters (and because packages are often viewed as modules and decoupling within a module is usually not an issue), we assume throughout the following that accessing and accessed types are in different packages, so that for the case of JAVA, we define the protocol of a type T , $\pi(T)$, as the *set of its non-static, public methods*. Note that this excludes public fields — if one feels uncomfortable with this, we argue that a field can always be replaced by a set of accessor methods.



Now the set of all subsets (the powerset \wp) of the protocol of a type, $\wp(\pi(T))$, is partially ordered by set inclusion. In fact, $\wp(\pi(T))$ is a bounded lattice with the empty protocol at the bottom and the protocol of T at the top, and with set intersection denoting the infimum and set union denoting the supremum. For instance, for a type with four public methods, we get the following Hasse diagram depicting the lattice:

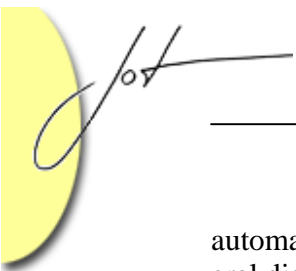


Generally, each node in such a lattice corresponds to a protocol of a type. However, owing to the usual placement of supertypes (with smaller protocols) above subtypes, we reverse the ordering and place the empty protocol at the top. Therefore, the bottom node has cardinality n , i.e., it comprises n (in the example 4) methods. One row above, we find all subsets of this protocol with cardinality $n-1$, two rows above those with cardinality $n-2$, and so forth. The node at the top corresponds to the empty protocol or that of the common root type (`Object` in JAVA¹). An edge between two nodes represents set (or protocol) inclusion: the protocol of the lower node includes all elements of the protocol of the upper one (note the reversal). Because set inclusion is transitive, edges spanning more than one level are not shown: each node in the graph “subsumes” (in the sense that its associated protocol includes the protocols of) all nodes reachable from that node by travelling edges upwards. In the same vein, every pair of nodes in the graph has a uniquely determined “greatest common subnode” (representing the least common superset of protocols) and a “least common supernode” (representing the greatest common subset).

Each node in the graph represents the protocol of a *structural supertype* of the types associated with the nodes reachable from that node by travelling edges downwards. In particular, all nodes in the graph represent the protocols of structural supertypes of the type represented by the bottom, which we call T_α (for *type under analysis*). In languages with structural type systems, a declaration element declared with T_α is assignment compatible with any declaration element declared with a type corresponding to any node in the lattice. Generally, structural subtyping is a prerequisite for substitutability, since a subtype’s protocol must include everything that is expected from its supertypes.

In languages with nominal type systems (JAVA etc.), subtyping must be declared explicitly. These languages usually ensure substitutability (structural subtyping) in presence of *declared* (nominal) subtyping by means of inheritance, i.e., by letting every subtype inherit the protocol of its nominal supertypes (so that protocol inclusion follows). Note that a type whose protocol happens to be a subset of the protocol of another type is not

¹ JAVA’s `Object` has twelve members, which are inherited by all types. We do not count these here.



automatically a declared (nominal) supertype of the other type; also, a type can have several distinct (distinctly named) declared subtypes and supertypes with identical protocol.

Thus, we have:

- T_α , the type under analysis, at the bottom of the lattice;
- *structural supertypes* of T_α , the types whose protocols are subsets of the protocol of T_α (all nodes of the lattice);
- *declared supertypes* of T_α , the nominal (and therefore also structural) supertypes of T_α ;
- *possible supertypes* of T_α , structural supertypes of T_α that are declared in the program, but are not declared to be supertypes of T_α (although they could be);
- *structural subtyping*, the edges between protocols related by set inclusion; and
- *nominal subtyping*, the subtype relation defined by `extends` and `implements` clauses.

In Section 3, we will show how we annotate lattices to differentiate declared and possible supertypes.

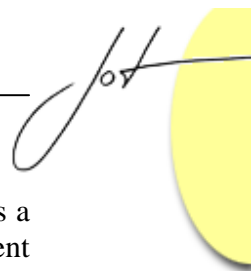
Type inference

Type inference computes type annotations from the structure of a program. It is usually applied to untyped declaration elements, inferring their types from their context. Because the typing information is extracted from the program, it is redundant with it so that its ability to detect logical programming errors is somewhat limited. However, it serves all other purposes of typing, namely the design of memory layout for variables and their values as well as the improvement of readability and performance of a program.

Languages like JAVA require that all declaration elements are type annotated. Type inference as described above is then mostly useful for closing dynamic loopholes, e.g. for making downcasts type safe [23]. More recently, type inference has also been offered as a convenience function for the programmer, who is freed from providing manual type declarations for temporary variables [11] or generic types [8]. By contrast, we are taking type inference in the opposite direction: since our goal is the extraction of interfaces (with the intent to decouple code), we are not interested in the inference of a missing type annotation or the narrowing of one that is not specific enough, but rather in a relaxation without losing type safety. This is based on our observation that the types used in declarations are often overly specific, i.e., they contain members not needed from the declared elements (variables etc.) so that coupling is unnecessarily strong [9, 16, 18].

In [18, 19], we have described the implementation of a type inference algorithm for JAVA that serves our purpose. In particular, it computes, based on a single selected declaration element, the exact protocol needed from its type, as written down in the program. For this, it analyses the use of the declaration element in its scope, as well as the use of all declaration elements it gets possibly assigned to. Note that assignment can be implicit, for instance in the parameter passing of method calls and returns.

Because of the static typing rules of JAVA, the protocol of the inferred type of a declaration element is always a subset of the protocol of its declared type. Thus, the protocol



of the inferred type of each declaration element declared with a given type T_α occupies a node in the supertype lattice of T_α . Since the protocol of an inferred type of an element declared of T_α represents an access pattern of T_α , we call this protocol an *access set* of T_α . Note that access sets always correspond to protocols of structural supertypes; they may also, but need not, correspond to protocols of declared (nominal) supertypes, including T_α itself.

Thus, we have

- *declaration elements*, i.e., variables (fields, formal parameters, and temporaries) as well as non-void method returns;
- *assignments*, i.e., the passing of the content of one declaration element to another;
- *declared types* of declaration elements, i.e., the types referenced in the declaration of the elements;
- *inferred types* of declaration elements, i.e., the types comprising only the methods actually needed from a declaration element and the ones it gets assigned to;
- *access sets*, i.e., protocols required from (inferred types of) declaration elements.

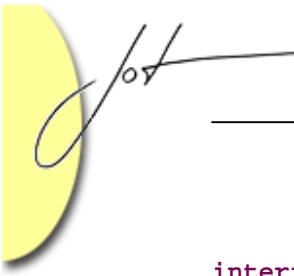
In the next section, we will show how this information can be combined with the supertype lattice to serve informed type refactoring of the program.

3 REFACTORING BASED ON DECLARED AND INFERRED TYPES

We illustrate our approach using a simple example. The code on which it is based, and which is shown in Figure 1, has been designed to be short and easily comprehensible, while at the same time showing some interesting problems and how our approach can help solve them in a straightforward manner. Many more realistic settings will be far more complicated, but this works in favour of our objective: the global constraints imposed on a program by a static type system such as JAVA's are hard to predict for a developer bogged down in the details of a class, and the more complex the program, the more the assistance we offer will be appreciated.

The story behind this example could be the following. A class `C` is used by a couple of clients. The developer wants to decouple `C` from `SuperClient` and `OtherClient`, by using appropriate interfaces in the clients' variable declarations (client/server interfaces [17]). For example, at first glance it seems that `I` could replace `C` in `SuperClient`. This however ignores the presence of `SubClient`, which the developer may easily overlook (especially if `SubClient` is developed by somebody else).

The supertype lattice of the type under analysis, `C`, is shown in Figure 2. Each node (marked with an N) occupied by a type is expanded to show the type name (marked by a green C or a purple I for class or interface, respectively; for the collapsed view in which the type name is not shown, the border is coloured correspondingly). Declared supertypes are distinguished from possible ones by drawing UML-style arrows (solid line and hollow arrowhead for *extends*, dashed line and hollow arrowhead for *implements*) where



```

interface I {
    void m();
}

interface J {
    void m();
    void o();
    void p();
}

class B {
    public void p() {}
}

class C
    extends B
    implements I {
    public void m() {}
    public void n() {}
    public void o() {}
}

class SuperClient {
    C cSC;
    void setServer(C c) {cSC = c;}
    void x() {cSC.m();}
}

class OtherClient {
    C cOC;
    void setServer(C c) {cOC = c;}
    void y() {cOC.m(); cOC.o();}
}

public class Main {
    public static void main(String[] args) {
        C cM = new C();
        SuperClient sc = new SuperClient();
        OtherClient oc = new OtherClient();
        sc.setServer(cM);
        oc.setServer(cM);
    }
}

class SubClient extends SuperClient {
    void x() {cSC.n();}
}

```

Figure 1: Source code of example (JAVA)

a subtype declaration exists. The subtyping of `Object` is, as usual, implicit and therefore not shown.

Except for the structural supertypes, the graph shows nothing but a diagrammatic form of the supertype hierarchy of `C`, which is already, albeit only in textual or treeview form, provided by the JDT in various places (for instance, in the supertype selection dialog of the GENERALIZE DECLARED TYPE refactoring).

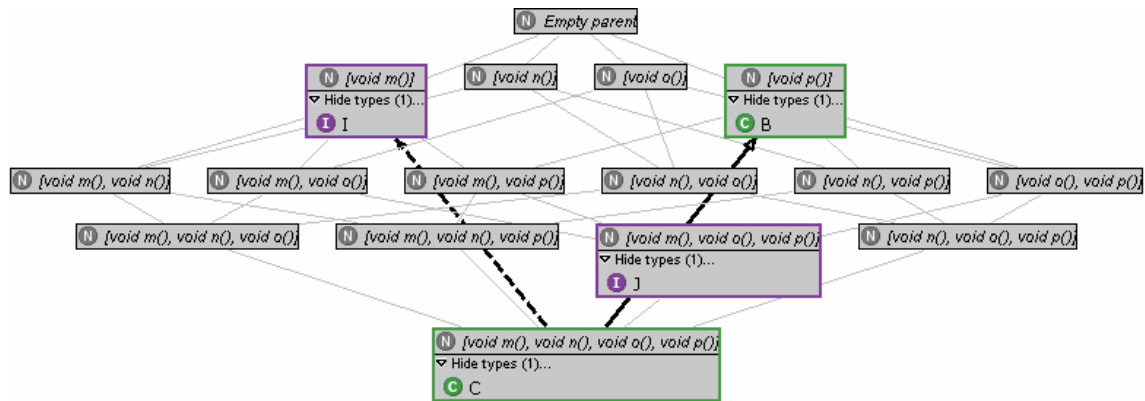
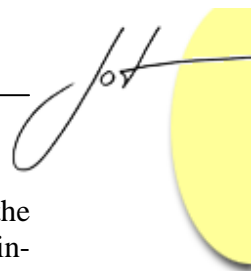


Figure 2



Based on the given type hierarchy and its visualization in the supertype lattice, the developer can perform a number of refactorings (although some only in a rather uninformed way):

1. *Extract new supertypes.* By selecting a node from the lattice and performing the corresponding refactoring, a new interface or (abstract) superclass of T_α can be introduced. The protocol associated with the node will then be the protocol of the new type. However, the graph provides no cues which supertypes might be useful, only which ones are already available.
2. *Review existing supertypes.* For an existing declared supertype containing too many or too few methods, the refactorings PULL UP/PUSH DOWN METHOD can be performed. Visually, this translates to dragging the supertype along structural subtyping edges to another node. Note that dragging the type down in the lattice corresponds to pulling up a method and vice versa. Also note that possible change is bounded by the existence of declared supertypes and subtypes, respectively; if a type were dragged beyond such a bound, the bound would have to be moved as well, or subtyping would have to be reversed and assignment compatibility between correspondingly typed variables would be lost. Last but not least, note that moving an existing supertype up in the lattice (amounting to pushing down or — in case of interfaces and abstract classes — dropping methods of its definition) is only possible if the removed methods are not used by any declaration elements in the program declared with that type. However, this information is currently not present in the graph, so that based on the given information alone, moving up cannot be limited accordingly. Moving a supertype down does not compromise its usability, since this only adds (unneeded) methods to the protocol (but see the possible problems noted in Section 4).
3. *Merge structurally identical supertypes.* Supertypes with same protocol, but different names, will occupy identical nodes in the graph. This can, but need not be, the case if one type subtypes another without adding methods (Family interfaces [17], for example). If the developer decides that these types express identical concepts, they can be merged into one. However, this introduces assignment compatibility of variables that were previously incompatible, so care must be taken that no unwanted use of program elements becomes possible by this. For instance, methods can be passed parameters of a type that was previously unacceptable. Also, joining previously distinct types may break method overloading (which is a strong indication that types were intentionally kept separate).
4. *Merge similar supertypes.* Structurally similar supertypes, especially cases in which one type subtypes another without adding much (as visualized by their being located on neighbouring levels and a correspondingly short path between them), can also be detected from the graph. In this case, the developer may decide to merge the types in question, by moving them to a common subnode. In fact, by selecting two or more nodes the greatest that comprises all is uniquely determined and can be marked in the lattice automatically. However, the limitations of reviewing supertypes (Refactoring 2) and merging identical supertypes (Refactoring 3) apply.

5. *Introduce subtype declarations.* An existing structural supertype that is not yet also a declared supertype can be made one, by introducing a corresponding edge (*extends* or *implements*). In the given example, *J* could be made a supertype of *C*. However, the objections regarding additional assignment compatibility made in the context of Refactoring 3 apply in full: more information is needed to be able to perform such a refactoring without loss of type safety.
6. *Use a supertype where possible.* Based on the awareness of the existence of declared supertypes, the corresponding refactoring can be triggered, with T_α as the source and the selected supertype as the target type. However, from the given graph it is not obvious how well a given supertype is already used, or whether it is at all usable (so that the refactoring will be successful).

We conclude that although a number of interesting refactorings are suggested by a supertype lattice, their applicability involves a certain amount of uncertainty. In the subsections that follow, we will reduce this uncertainty step by step.

Adding declarations

A move in this direction is the association of each type in the lattice with the set of declaration elements in the program declared with that type. The developer can then check whether and where each type is actually used. We therefore add the elements of a program declared of a type to the node of that type and colour all nodes possessing declaration elements green, as shown in Figure 3. The information so obtained can help decide to

- perform a USE SUPERTYPE WHERE POSSIBLE refactoring (Refactoring 6) for a supertype that is not used as much as expected; to
- refactor a supertype that turned out to be not as usable as expected (because applying USE SUPERTYPE WHERE POSSIBLE on this type did not change the number of declaration elements in the graph in a satisfactory manner); or to

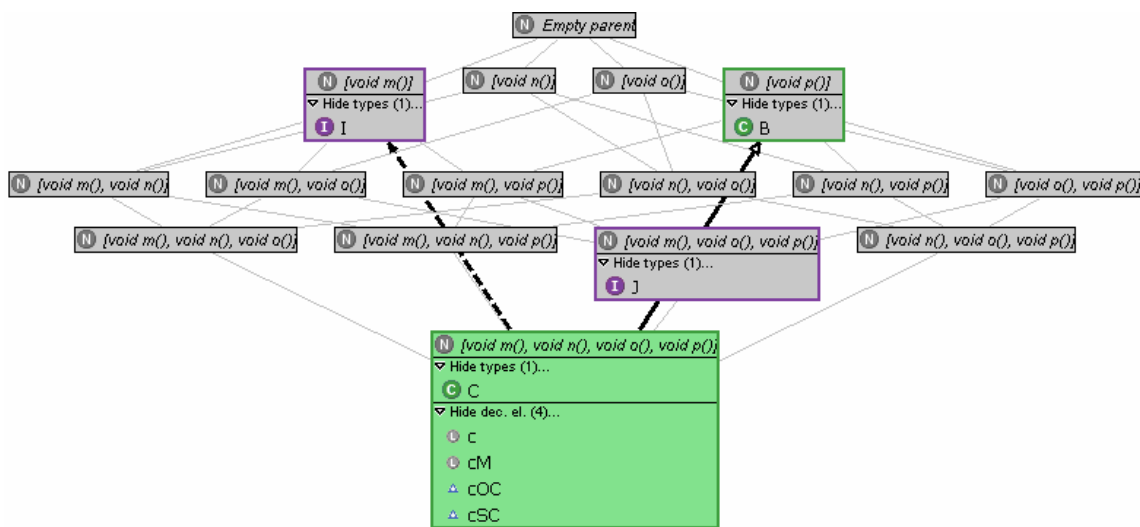


Figure 3

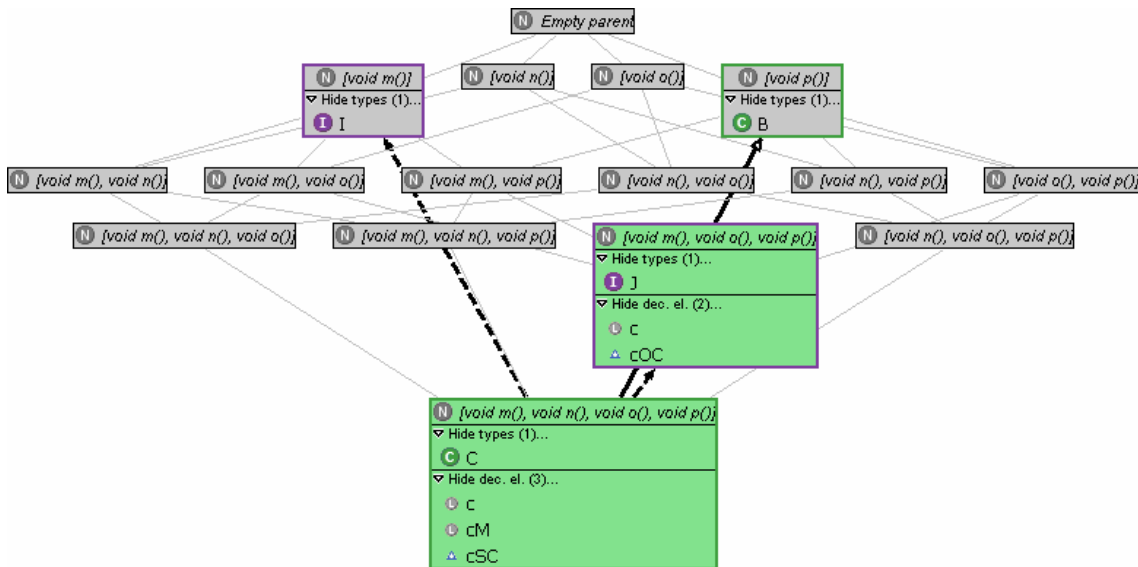


Figure 4

- introduce a new supertype (Refactoring 1) because the ones already existing, although rarely used, must not be touched.

Refactoring a supertype can involve pulling up or pushing down methods as described in Refactoring 2, merging types as described in Refactorings 3 and 4, introducing subtype declarations as described in Refactoring 5, or simply deleting a supertype that is unusable. However, all these refactorings have to be performed rather blindly, that is, in a trial and error fashion (the preview function of USE SUPERTYPE WHERE POSSIBLE can be used to see whether and where a type is usable, but this works only for types as are and is extremely cumbersome).

In case of our example, we assume that the developer decides that interfaces **I** and **J** should be used to decouple clients from **C**. Since **C** does not subtype **J** and **J** is not used anywhere (so that no unwanted assignments can result; cf. above), the first step is to introduce a corresponding subtype relationship (Refactoring 5). The next step is to replace occurrences of **C** in **SuperClient** and **OtherClient**, which is routinely done by applying USE SUPERTYPE WHERE POSSIBLE to **C** with **I** and **J** as supertypes, respectively. However, of these refactorings only one is successful: as can be seen from Figure 4, **C** has been replaced by **J** in **OtherClient**, but **I** could not be used anywhere in the program.²

Unfortunately, the refactoring provides no hints as to why this is so; it is up to the developer to inspect the code and detect that **SuperClient**'s method **x** is overridden in **SubClient**, invoking method **n** of **C** which is not offered in **I**. However, as will be seen next, this information can be derived automatically from the program.

² In the version of ECLIPSE we are currently using (3.2.1) USE SUPERTYPE WHERE POSSIBLE actually does use **I** in the declaration of field **c** in **SuperClient**, introducing a typing error (Bug# 171950). Conversely, USE SUPERTYPE WHERE POSSIBLE does not suggest the use of **J** instead of **C** in **OtherClient** (Bug #171949). However, GENERALIZE DECLARED TYPE works correctly in both cases, the drawback being that it has to be applied for every declaration element separately (and in the right order).

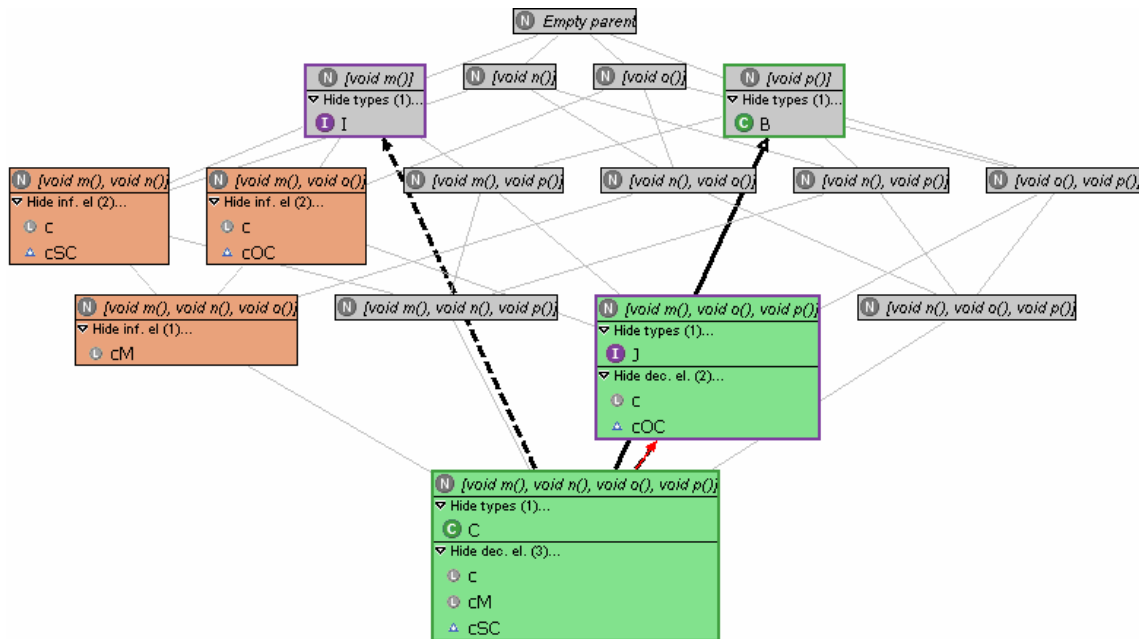
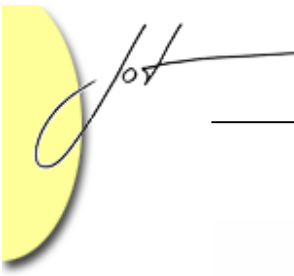


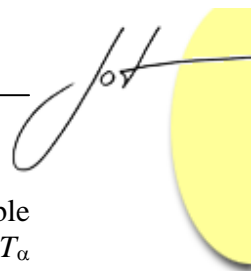
Figure 5

Exploiting access sets

With the type inferring (access set collecting) algorithm mentioned in Section 2 at hand, we can further enrich the above supertype lattice, by counting for and associating with every node the declaration elements having the precise access set (protocol of the elements' inferred type; see Section 2) represented by the node. Applied to our example, adding type access information produces the graph of Figure 5 (note the new compartment containing the declaration elements whose inferred type falls onto the node; nodes that represent such access sets are coloured red; types for which an assignment exists are additionally connected through a red arrow).

Now attempting to redeclare an element with a type that is not a subtype of its inferred type (i.e., that is not connected through structural subtyping edges in the graph) introduces a typing error, since this type misses needed methods. Therefore, information provided by the supertype lattice enhanced as above extends that available previously by showing the developer for which declaration elements Refactoring 6 (USE SUPERTYPE WHERE POSSIBLE) with a selected supertype will not produce any changes, namely for those elements for which no path from the node representing the protocol of their current declared type to that representing their access set visits the node representing the selected type. Also, the information provided by the access sets supports refactorings in the following ways:

- A relatively often used access set suggests the creation of a corresponding new supertype of T_α (Refactoring 1). Also, if one access set includes several others that together represent frequent (as compared to that of other access sets) use, that



access set is a good candidate for a new type. The new type will likely be usable for many declaration elements having the access set and currently declared of T_α (Refactoring 6), but not necessarily for all; especially if a declaration element gets assigned to a supertype of its declared type and if that type is not also a (structural) supertype of the new type, redeclaring this element requires further measures.³

- A rarely used supertype subsumed by (i.e., residing above) a frequently used access set suggests that this type lacks the additional methods the access set requires. Moving the supertype down to the corresponding node (Refactoring 2) likely makes it more usable.
- Conversely, that the full protocol of the supertype is not used at all (as indicated by no access set coinciding with the protocol of the type) suggests that it may be moved up, to the node representing the least common superset (note again the reverse order mentioned in Section 2) of all access sets placed above it, if that node is different from that of the supertype (Refactoring 2). On the other hand, that it is used does not mean that it cannot be moved up, simply because the access set can be that of a declaration element typed with a subtype.
- A supertype whose associated node subsumes those of frequently used access sets and which is used in few declarations (as can be seen from a small number of declaration elements using it) is a good candidate for a USE SUPERTYPE WHERE POSSIBLE refactoring (Refactoring 6).

Applied to our concrete example, the following observations can be made, giving rise to the corresponding refactorings:

- Interface **I** is not used at all, but resides above an access set used twice (which is frequent in our small example). Pulling up the missing method, namely method `void n()`, from **C** to **I** moves **I** down to the access set (Refactoring 2), so that USE SUPERTYPE WHERE POSSIBLE can retype the two declaration elements having this access set with **I** (Refactoring 6).⁴ But these are precisely the two declarations of `SuperClient` using **C**, so that `SuperClient` and `SubClient` are now decoupled from **C** (do not reference **C** anymore).
- The node associated with interface **J** represents no access set, meaning that there is not a single declaration element that uses all of **J**'s protocol. On the other hand, there is a node with an access set used twice, which it subsumes. One should therefore consider dropping the superfluous method(s), in this case method `void p()`, from **J**. Although this might be rejected by the type system (because there

³ Cf. [19] for a more detailed analysis of the requirements for retyping with maximally general types. Note that for the new supertype to be usable in place of other types of the lattice, corresponding subtype declarations must be added to these types (Refactoring 5). Also note that if the new type is to be used for declaration elements that get assigned to supertypes, the new type must subtype these supertypes, or the assignments will be rejected by the type checker. Graphically, this corresponds to the insertion of the new type in the assignment chain, which necessitates corresponding subtyping links.

⁴ Once again, the USE SUPERTYPE WHERE POSSIBLE refactoring of ECLIPSE 3.2.1 does not do the job, but GENERALIZE DECLARED TYPE does.

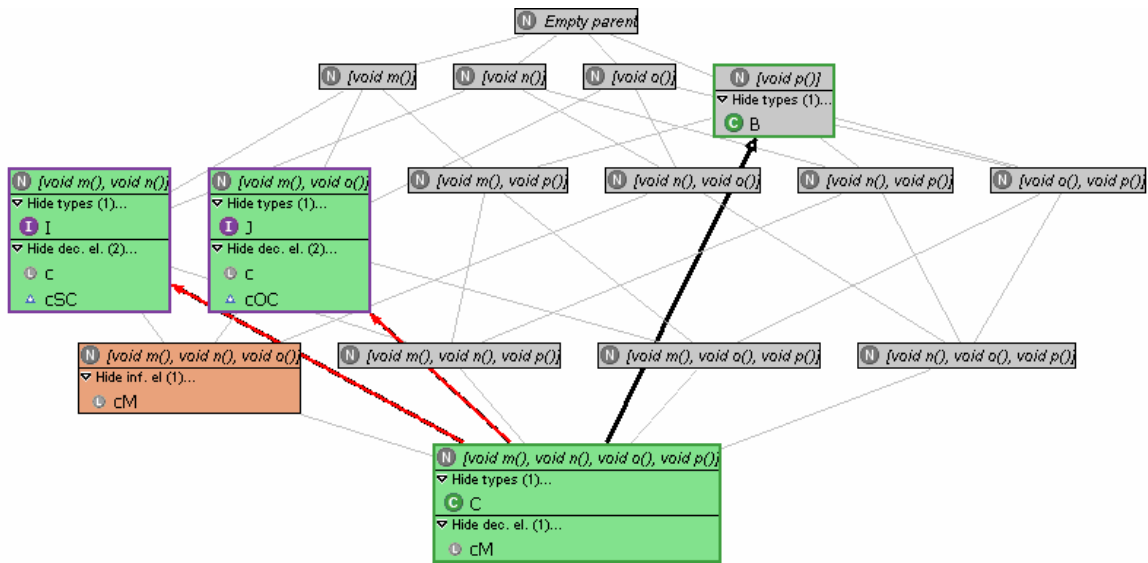
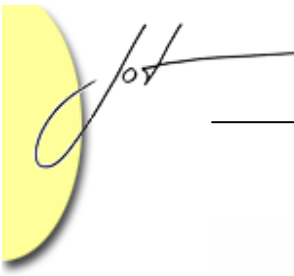


Figure 6

may be other elements declared of \mathcal{J} that need p ; cf. Section 4), in our example this is not the case.

The result is shown in Figure 6. Only a single declaration element (c_M) remains typed with C . However, even this declaration element does not make use of the full protocol of C — rather, its associated access set (which lacks p) is the least common superset of the protocols of I and J (in the above diagram represented by the reddish node below them). Because this access set includes all others in the graph, we can reduce visibility of p to non-public. In the given example, we can even go one step further and remove the `extends B` declaration from C (which lets C inherit p), since C is never used as a B (as indicated by the missing assignment link). This leads to the graph shown in Figure 7. Note that all performed refactorings were suggested by the annotated supertype lattice of C , and did not require manual inspection of the source code.

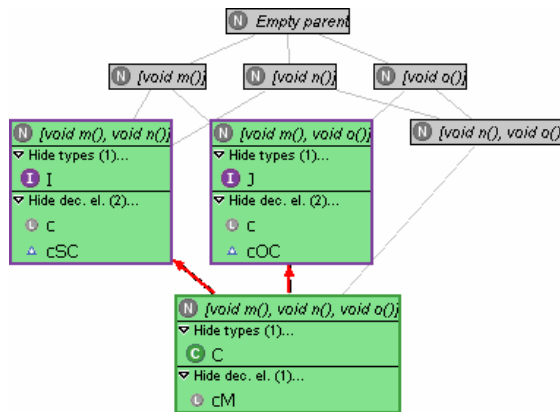
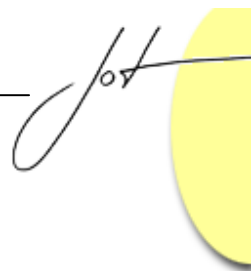


Figure 7



4 PRAGMATICS AND TOOL SUPPORT

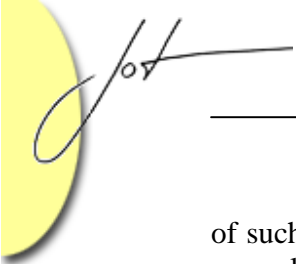
There are several practical requirements and limitations to our approach. First and foremost, with growing protocols of types the supertype lattice quickly becomes unwieldy (it grows to the power of 2). Therefore, in most practical cases not all nodes of the lattice will be shown, but only the ones that hold information. Earlier experiments of ours have shown that even for popular types (i.e., types used in many declarations) with protocols with more than 50 methods, the number of nodes obtained this way remains manageable. Alternatively, we could let the developer select from the protocol of the analysed type the methods for which access analysis is to be performed — those that have not been selected will then not be available for refactoring.

Another current problem is performance. For large type hierarchies with many declaration elements, type inference required takes its toll. Although we have not yet measured performance systematically and on a large scale, measurements with small projects indicate that computation times will be acceptable. For instance, with our current implementation of type inference type access analysis for the 40 types and 161 declaration elements of JUnit 3.8.1 takes less than 4 minutes on a contemporary PC, or less than 6 seconds per type. We are currently re-implementing our type inference algorithm using ECLIPSE's type constraint framework [8, 22], which may speed up the process considerably.

Of a different nature is the problem that type access analysis of one class may interfere with that of others, namely when the class's declared (nominal) supertypes are also supertypes of other classes not included in the lattice (because they are not supertypes of the first class). This has several consequences:

- Declaration elements with such a supertype may actually represent objects of classes outside the lattice. This can be avoided by including only declaration elements that can get assigned instances of T_α and its superclasses (as can be computed by following assignment arrows as shown in Figures 5–7).
- Changes to such a supertype (adding or deleting methods) that are OK for the subtypes contained in the lattice may also affect types outside the lattice, for which the access analysis provides no information. Such changes may induce typing errors, which must be prevented by additional constraints not described in this paper.

Another limitation of our work is the occurrence of typecasts. Because of the typing rules of JAVA and similar languages, assignments are possible only from subtypes to supertypes. Therefore, all assignment edges in the lattice are directed from bottom to top (and are backed up by the existence of a path from the corresponding lower to the upper node). However, in JAVA expressions may be downcast — or crosscast — before an assignment, information that could also be included in the graph. In order not to further complicate matters, we did not pursue this further and assume that every typecast in a program represents an explicit statement of a developer that the type that is being cast to is the type that is actually wanted, and that this type is not to be replaced (as is the case for constructor calls). However, not considering typecasts may lead to clashes when changing, merging, or deleting a type that is the target of a cast. Currently, the detection



of such errors is left to the compiler; the next, constraint-based version of our type inference algorithm will contain a switch that allows the consideration of casts and constructor calls as if they were declaration elements.

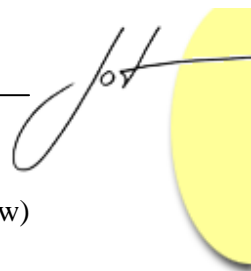
There is also a number of more technical issues. For instance, the introduction and use of new supertypes for the (formal) parameters of methods may turn unambiguous method calls into ambiguous ones. This is a known problem for many refactorings (see, e.g., [22]); however, during the rigorous testing of our INFER TYPE refactoring tool [18, 19] this problem never occurred. Next, type inference as described in [18, 19] and the type system of JAVA may require the introduction of an abstract class instead of an interface, or of more than one new type in certain cases; while the former is naturally covered by the supertype lattice, the latter requirement is more difficult to integrate.

We have implemented the type access analysis described in this paper as an ECLIPSE plug-in, called TAA (for TYPE ACCESS ANALYSER). The figures shown in this paper are screenshots from this tool. In addition to what can be seen here, all elements contained in the graph are linked to the analysed program source, from which the refactorings can be invoked. For instance, by hovering the mouse pointer over a declaration element in the graph one gets its complete signature as a tool tip annotation; double clicking on the element opens the element inside a JAVA source file editor, where the refactoring can be performed as usual.

5 RELATED WORK

Systematic identification and definition of refactorings, as pioneered by Opdyke & Johnson [13] and continued in Fowler's seminal book [7], is still an area of active research and development. Parallel to this, work on what has been termed "bad smell detection", i.e., on the identification of locations in the source code where a certain refactoring should be applied, has been commenced [2]. In this paper, we pursue neither direction: instead, we present the theory and a tool that help maximize the effect of type-related refactorings, once the developer has decided that such a refactoring should be performed.

The automated refactoring of class hierarchies is well studied and by now has a long history. In fact, one of the first published works on refactoring deals with the introduction of abstract superclasses [13]. [6, 12] give good overviews of automatic class insertion algorithms, the former also in presence of overloading (a constraint that is often ignored [19, 22]). Other, more recent work deals with the refactoring towards use of existing types [1, 8, 22]. Our work, by contrast, collects all information from a program that is relevant for the design and use of generalizations (abstract classes and interfaces), and presents it to the developer in such a way that it can assist her/him with the refactorings she/he wants to make. We believe that such support is superior to all automated refactorings of type hierarchies and redeclarations of variables of the above-mentioned kind, mostly because developers are likely to perceive automatic refactoring (even though it may be eye-opening in certain cases) as dull, simply because it ignores their intentions.



We respect the director role of the developer and offer the opportunity to design a (new) type as suggested by our type access analysis, and to use it where deemed appropriate.

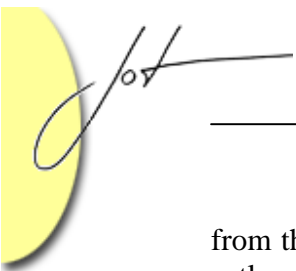
To our knowledge, the only direct competitor of our work is Streckenbach & Snelting's system KABA [20, 21]. KABA builds on prior work by Snelting & Tip on formal concept analysis [15], which allows for a given program the computation of a new, functionally equivalent type hierarchy containing minimized types (viz. classes) only. Although at first glance the goals of KABA and our TAA appear to be the same, namely to suggest refactorings that tailor a type hierarchy to the use by its clients, there are fundamental differences, both theoretical and technical in nature. First and foremost, the goal of KABA is to minimize classes, that is, to instantiate every object of a system from a class that has all and only the members that are needed from that object. For this purpose, KABA specializes each analysed class into a number of subclasses, one per distinct use, and pushes down members as far as possible. The result is smaller memory footprint for objects, but each client of a class is now bound specifically to a certain subclass of it, and mutual substitutability of objects that once belonged to the same class is lost, making the whole approach extremely sensitive to small changes (a phenomenon already noted in [14]). By contrast, we do not specialize classes, but rather create abstractions (which cannot be instantiated) that serve the decoupling of clients from the original class and its subclasses. In particular, instance creation remains unaffected by our approach. To paraphrase, KABA minimizes objects, while we minimize variables (minimize in the sense of removing superfluous members). This is also reflected in the technical groundwork: KABA uses points-to analysis, while we use class hierarchy analysis [5].

Another significant difference between KABA and our TAA concerns usability: while KABA operates on bytecode and provides a refactoring tool that is separate from the IDE in which the code to be refactored was created, our TAA plugs into ECLIPSE and works side by side with all other ECLIPSE refactorings. In fact, as stated above it is not itself a refactoring tool, but presents a structure on which existing refactorings can be performed in an informed manner.

6 FUTURE WORK

Our work can be continued in a number of ways. One is to provide the possibility of switching from aggregate to detailed information, for instance for following the assignments starting with a certain declaration element. Such could be used to harmonize the naming of variables in an assignment chain or, in case of a variable name change beyond a certain point, serve as indication for the introduction of a new interface (reflecting the new role of the objects passed to that variable).

Another interesting extension would be to highlight in the supertype lattice all and only the declaration elements that are declared in a certain class or package. This would allow one to investigate the perspective this class or package has on the type under analysis, i.e., how often it references this type and how each reference/all references are used. Based on this view, a single interface could be designed that decouples the class/package



from the type and that could be used in the import section of the class(es) to specify exactly what is required from that type (the “required interface”). In this context, the USE SUPERTYPE WHERE POSSIBLE refactoring could be confined to change declaration elements of the requiring class/package only.

The information obtained by our type access analysis can also be used to trigger other refactorings. For instance, presence of subtyping without making use of substitutability (i.e., without presence of an assignment chain from the subtype to the supertype) hints at the fact that the subtyping was introduced only for the purpose of inheritance. Especially if further analysis shows that only few of the inherited methods are actually used, it may be better to “replace inheritance with delegation” [7]. But even if there are assignments, this refactoring can be applied, given that the superclass from which is inherited itself has an (abstract) supertype and that the protocol of this supertype coincides with or includes all access sets determined for the inheriting type. The delegator would then subtype this supertype (rather than inherit from the former superclass), and the former superclass would become the delegate.

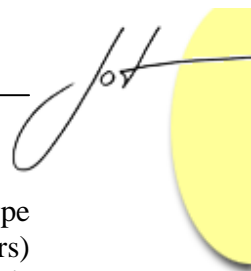
Last but not least, an interesting extension would be to offer a REDUCE VISIBILITY refactoring that is based on type access analysis. If, for a selected T_α , the least common superset of all access sets does not coincide with the protocol of T_α , then the visibility of all members not contained in this node can be lowered, at least from public to protected. Whether it can be lowered further brings us back to the problem of relativity of visibility in JAVA mentioned at the beginning of Section 2; to be able to decide this, our type inference algorithm would have to be changed to consider other than public visibility, which we expect to be doable, but non-trivial. However, it should be clear that the type access lattice as described here cannot be extended to cover graded accessibility in a straightforward manner: because membership in a protocol is no longer binary, it cannot be built on set inclusion of protocols as partial ordering; also, structural subtyping depends on locations and is no longer antisymmetric (e.g., two types differing only in their private members are mutual structural subtypes without being identical).

7 CONCLUSION

Type hierarchies are inherently difficult to refactor. While introducing a new supertype is simple, using it in the declarations of a program requires knowledge of

- the use of the elements to be declared with it (in terms of what is expected from, or accessed on, the objects it represents), and of
- the existing assignments the elements are involved in (so that the new type has the super- and subtypes required by the language’s rules of assignment compatibility).

Similarly, changing an existing type by adding or removing members requires knowledge of its super- and subtypes, and of what is accessed on the elements of the program already declared with that type. With our TYPE ACCESS ANALYSER implemented as an ECLIPSE plug-in, we believe to have provided assistance to the developer that was previously unavailable: changes to the type hierarchy that may improve the modularity of a program



(by decreasing its inherent coupling) are suggested visually as annotations to a supertype lattice, as are limits to changes of a supertype (in terms of adding or deleting members) that are possible without other alterations of a program. The extension of our tool in various ways follows naturally.

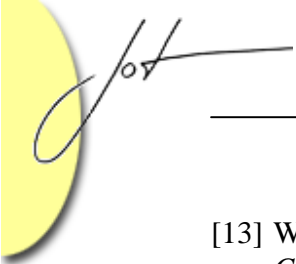
ACKNOWLEDGMENTS

The authors are indebted to Andreas Meißner for implementing the original TAA display (from which the raw lattice in Section 2 was taken) and for helping with the intricacies of GEF. Also, the anonymous referees were exceptionally helpful (especially reviewer #4).

This work has been partially sponsored by the EU project SENSORIA, IST-2005-016004.

REFERENCES

- [1] I Balaban, F Tip, RM Fuhrer “Refactoring support for class library migration” in: *Proc. of OOPSLA* (2005) 265–279.
- [2] A Bhattacharya, RM Fuhrer “Smell detection for Eclipse” in: *Proc. of OOPSLA Companion* (2004) 22.
- [3] G Bracha, D Griswold “Strongtalk: Typechecking Smalltalk in a production environment” in: *Proc. of OOPSLA* (1993) 215–230.
- [4] PS Canning, WR Cook, WL Hill, WG Olthoff “Interfaces for strongly-typed object-oriented programming” in: *Proc. of OOPSLA* (1989) 457–467.
- [5] J Dean, D Grove, C Chambers “Optimization of object-oriented programs using static class hierarchy analysis” in: *Proc. of ECOOP* (1995) 77–101.
- [6] H Dicky, C Dony, M Huchard, T Libourel “On automatic class insertion with overloading” in: *Proc. of OOPSLA* (1996) 251–267.
- [7] M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).
- [8] RM Fuhrer, F Tip, A Kiezun, J Dolby, M Keller “Efficiently refactoring Java applications to use generic libraries” in: *Proc. of ECOOP* (2005) 71–96.
- [9] J Gößner, P Mayer, F Steimann “Interface utilization in the JAVA Development Kit” in: *Proc. of SAC 2004* (ACM, 2004) 1310–1315.
- [10] *Infer Type documentation* (<http://www.fernuni-hagen.de/ps/prjs/InferType>).
- [11] Microsoft *C# Version 3.0 Specification* (<http://msdn.microsoft.com/>).
- [12] I Moore “Automatic inheritance hierarchy restructuring and method refactoring” in: *Proc. of OOPSLA* (1996) 235–250.

- 
-
- [13] WF Opdyke, RE Johnson “Creating abstract superclasses by refactoring” in: *ACM Conf. on Computer Science* (1993) 66–73.
- [14] J Palsberg, MI Schwartzbach “Object-oriented type inference” in: *Proc. of OOPSLA* (1991) 146–161.
- [15] G Snelting, F Tip “Understanding class hierarchies using concept analysis” *ACM TOPLAS* 22:3 (2000) 540–582.
- [16] F Steimann, W Siberski, T Kühne “Towards the systematic use of interfaces in Java programming” in: *Proc. of PPPJ* (ACM, 2003) 13–17.
- [17] F Steimann, P Mayer “Patterns of interface-based programming” *Journal of Object Technology* 4:5 (2005) 75–94.
- [18] F Steimann, P Mayer, A Meißner “Decoupling classes with inferred interfaces” in: *Proc. of ACM Symposium on Applied Computing* (2006) 1404–1408. Extended version appeared as [19].
- [19] F Steimann “The *Infer Type* refactoring and its use for interface-based programming” *Journal of Object Technology* 6:2 (2007) 67–89.
- [20] M Streckenbach, G Snelting “Refactoring class hierarchies with KABA” in: *Proc. of OOPSLA* (2004) 315–330.
- [21] M Streckenbach *KABA — A System for Refactoring Java Programs* (PhD thesis, Universität Passau, 2005).
- [22] F Tip, A Kiezun, D Bäumler “Refactoring for generalization using type constraints” in: *Proc. of OOPSLA* (2003) 13–26.
- [23] T Wang, SF Smith “Precise constraint-based type inference for JAVA” in: *Proc. of ECOOP* (2001) 99–117.

About the authors



Friedrich Steimann is a full professor for Programming Systems at the Fernuniversität in Hagen, Germany. He leads a research group on object-oriented languages, software modelling, and programmers’ productivity tools. He can be reached as steimann@acm.org.



Philip Mayer is a research assistant at the Department of Programming and Software Engineering of the Ludwig-Maximilians-Universität München, Germany. His research interests include program understanding, software testing, and component-oriented architectures. He can be reached at plmayer@acm.org.