JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 2, Special Issue OOPS Track at SAC 2006, February 2007

The *Infer Type* Refactoring and its Use for Interface-Based Programming

Friedrich Steimann, Fernuniversität in Hagen, Germany

Abstract

Interface-based programming, i.e. the systematic use of interface types in variable declarations, serves the decoupling of classes and increases a program's changeability. To maximize this effect, interfaces should contain as few elements as possible. For the design of minimal (i.e., maximally general) interfaces, an in-depth analysis of the protocol needed from objects in a given context is required. However, currently available refactorings for the creation of such interfaces (such as *Extract Interface*) leave programmers alone with the decision what to include or, more importantly, what to omit: they let them choose manually from the protocol of a class, and only then offer the use of the new interface where (if) possible. To end this trial and error process, we have developed a new refactoring named *Infer Type* that, using type inference, completely automates the construction of new, context-specific interfaces and their use in variable declarations, thus supporting greater decoupling and access protection of code, serving the goals of interface-based programming.

1 INTRODUCTION

Interface-based programming is an object-oriented programming technique by which object references (variables and methods with return values) are statically typed with abstract interfaces rather than concrete classes. These interfaces are typically context-specific [Steimann & Mayer 2005], i.e., they contain only the protocol needed from the referenced objects as accessed through the references setting up the context. Two main benefits are expected from this:

- it *restricts the access* to objects according to the individual accessor's perspective and thus goes beyond the possibilities of the usual access modifiers which — apart perhaps from a package local declaration — cannot address, or be tailored for, specific clients; and
- it opens up the code for *greater flexibility*, by reducing the dependency (measured in terms of the size of the required protocol) to what is actually needed rather than what happens to be offered by an available type, thus increasing plugability.

Cite this article as follows: Friedrich Steimann: "The *Infer Type* Refactoring and its Use for Interface-Based Programming", in *Journal of Object Technology*, vol. 6, no. 2, Special Issue OOPS Track at SAC 2006, February 2007, pp. 99-120 http://www.jot.fm/issues/issue_2007_01/article5.

As it turns out, both issues are of little concern for closed, monolithic applications, all parts of which are designed to go together so that the usual private/public access modifiers of class members suffice. They become increasingly more interesting, though, once applications are generalized into product lines, and are of utmost importance for the design of object-oriented application frameworks.

Although the interface-as-type construct, which was first proposed in [Canning et al. 1989], has been part of the JAVA language specification right from its beginning, interface abstractions for two of the JAVA API's most popular classes, `Vector` and `String` [Gößner et al. 2004], were only introduced in versions 1.2 (`List`) and 1.4 (`CharSequence`), respectively; not before its latest version (1.5) was `Iterable` added as a common interface for collections and other iterable objects. As it turns out, Sun's slowness in adopting its own interface construct is indicative of a certain programming style [Steimann et al. 2003, Gößner et al. 2004]; in fact, support for interface-based programming is stronger in C# than in JAVA, mirroring Microsoft's known preference of interface over implementation inheritance [Pattison 2000]. A more detailed account of the historical development of interfaces and their use can be found in [Steimann & Mayer 2005].

One of the major obstacles to interface-based programming is the problem of deciding precisely what to put in a new interface. If insecure about this, it is tempting to use an available type (a class usually) instead. While this may be acceptable as a workaround, the use of the class should later be refactored to that of a more context-specific abstraction. For this purpose, we have devised a new refactoring named *Infer Type* that, when applied on a program's variable or other object reference, computes the most general type (i.e., the one with the smallest protocol) that can be used for this reference (and possibly others it gets assigned to) without type casts. The algorithm underlying *Infer Type* is also used to compute a set of metrics guiding the programmer in whether the introduction is justified, and what its effect is on the decoupling of a program. *Infer Type* complements existing refactorings such as *Use Supertype Where Possible* and *Generalize Type* [Tip et al. 2003] and provides an alternative for the current implementations of *Extract Interface* [Fowler 1999], which perform no such analyses.

The remainder of this article is organized as follows. First we set the background of our work by giving a set of motivating examples. Then we present a few definitions necessary for the specification of our metrics and refactorings. In Section 4 we describe some of our metrics in greater detail, indicating how they can be employed to point to possible application of *Infer Type*, whose type inference algorithm is described in Section 5. In Section 6 we show how the refactoring functions as an ECLIPSE plugin. Section 7 provides some numbers reflecting on the cost of the systematic use of inferred types; they have been obtained through the automatic application of *Infer Type* to all references of two middle-size projects. We then discuss our work in light of certain language-specific and other problems (Section 8), and compare it with the work of others (Section 9). Suggestions for future work and a conclusion finish our contribution.

2 MOTIVATION OF OUR WORK

An interface abstracts from its implementing classes by factoring out the common public protocol of their use (ideally — although today often lacking the technical possibilities — including formal behaviour specifications), *hiding the realization* of the protocol. If not a complete (total) interface of its implementing classes, it also abstracts by *omitting certain public features* from their protocol, usually those features that are not needed in the context in which the interface is to be used (a *context-specific interface* [Steimann & Mayer 2005]). In this case, the interface is also a (true) generalization of its implementing classes. Figure 1 shows two versions of a program fragment of which (a) does not use a (context-specific) interface, while (b) does.

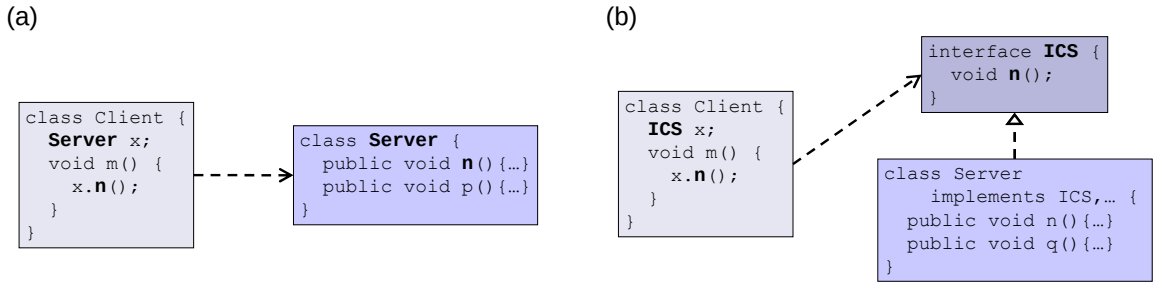


Figure 1: (a) Conventional programming without using interfaces in variable declarations. (b) Refactored code with an interface type that is specifically designed to suit variable `x`'s need. The overlaid UML class diagrams show dependency and subtyping in graphical form.

The method declarations collected by an interface usually come with formal parameters, which are also typed. In pure interface-based programming, these parameters should also be typed with interfaces rather than classes. This allows a user of a class to remain ignorant not only of its exact type, but also of those of its parameters, fields etc., further increasing the decoupling.

Requiring that all variables be typed with interfaces, context-specific ones especially, is certainly too strong a notion. In practice, the degree to which the maxim of interface-based programming is attained is limited by

- the maintainability of the resultant type hierarchy and
- the nature of the variables in question.

As for the former: the larger the type hierarchy, the harder it is to comprehend and to maintain, which is clearly at odds with the most fundamental design goals. In particular, it is usually not considered good design if every change in a method (be it in its declaration or its implementation) requires some restructuring of the type hierarchy (the *sensitivity to small changes* noted in [Palsberg & Schwartzbach 1991]). As for the latter, we observe that quite often it is natural to reference a class directly, and introducing an interface for this reference just makes no sense. This is for instance the case with object qualities, or attributes, (often implemented by primitive types such as `int` or `String`) and internal object composition (aggregation, realized by private fields). However, such choices depend on the intentions of the programmer and cannot be decided automatically; in par-

ticular, it is not useful to introduce interfaces just because it can be done — rather, there must be a concrete expected benefit, such as access protection or a future variation point in the design.

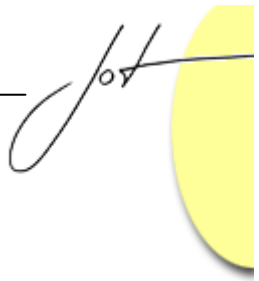
The following examples shed some light on these issues.

Example 1. Suppose a component or framework accepts a collection of objects and offers to treat them in a certain way. For this purpose, it publishes a method `void treat(Collection c)`. This however binds (couples) the component's clients not only to the component, but also to type `Collection`, since it accepts only parameters of this type or some subtype thereof.

Now suppose that closer inspection of the component reveals that all it does with a collection (viz. the objects referred to through formal parameter `c`) is to obtain an iterator so it can iterate over the collection's elements. Since non-collections may also provide iterators, the interface of the component unduly restricts its use — it is not as general as it could be. Applying *Infer Type* on parameter `c` would produce the interface `{Iterator iterator();}` which could be named `Iterable`, exported by the module, and replace `Collection` in its interface. Users of the component are then free to pass any actual parameter type they like as long as it is iterable. For this, the client program's own classes simply have to implement the new interface; other libraries' types (including the original `Collection`) can be subtyped and the new subtype made to implement `Iterable` (*cf.* [Läufer et al. 2000] for a discussion of retroactive type abstraction). ♦

Generally, the use of types external to a module in the interface of that module (for instance from a lower level or base library) brakes a layered design, because dependencies are passed through from lower to higher levels. Often this is not conceived as a problem, since the base library (for instance the JAVA API) is thought of as an integral part of the programming language (which is actually the case for types such as `Object`, `String`, and `Exception`), and almost no program works without this library so that the dependency exists anyway. However, this need not generally be the case; instead, it makes sense to let a module (or layer) come with its own parameter types. The client can then import (establish a dependency on) the base library if it chooses to, or else use its own types, or types from a different library.

Example 2. Suppose a client turns to a server for some specific service. In order to deliver this service, the server requires certain information from the client, precisely which depending on the state of the server or its environment. Rather than passing all the information that is possibly needed with the service request (method invocation), the client makes itself known to the server (by passing `this` as a parameter) so that the server can call back the client for whatever it needs to know. The following JAVA code gives a brief example of this:



```
class Client {
   Server s;
   Client() {s = new Server();}
   void do() {s.doThis(); s.doThat(this);}
   void help() {…}
}

class Server {
   void doThis() {…}
   void doThat(Client c) {… c.help(); …}
   void doSomethingCompletelyUnrelated() {…}
}
```

Now well-factored client/server type relationships are usually *m*:*n*. While in the given case it is obvious how to abstract from different servers (simply by introducing an interface declaring the service in question and letting all servers implement it), it is less trivial to find an abstraction of the different clients, since what the servers need from them is buried in the servers' code and — possibly — that of their delegates. Inferring the type of the server's formal interface parameter `c` (on which the callback is to be issued) does exactly this. Refactored the above code becomes the following:

```
interface IClient {void help();}

interface IServer {void doThis(); void doThat(IClient c);}

class Client implements IClient {
   IServer s;
   Client(IServer s) {this.s = s;}
   void do() {s.doThis(); s.doThat(this);}
   void help() {…}
}

class Server implements IServer {
   void doThis() {…}
   void doThat(IClient c) {… c.help(); …}
   void doSomethingCompletelyUnrelated() {…}
}
```

Note that this is the very situation encountered in many instances of the Observer pattern [Gamma et al. 1995]. ♦

Example 3. The JAVA API somewhat unorthodoxly defines `Stack` as subtype of `Vector`, allowing its users to pass a stack wherever a vector is required. This risks a stack being used by other code (to which it is passed as an actual parameter) in such a way that the specification of the stack is broken, for instance by inserting or removing elements at arbitrary positions. By inferring the type on the formal parameter (with declared type `Vector`) of the method in question one can check — without working through the code — whether non-stack methods are being accessed. ♦

Example 4. Reengineering legacy code may require the identification of UML-style collaborations. The participants of such a collaboration (formerly called *collaboration roles*) are protocol specifications, or interfaces [Steimann 2001]. In order to extract those interfaces, one has to identify the links (as represented by variables) underlying the collaboration. Inferring their types yields the different roles played by each collaborator, which together specify the requirements (required protocol) of a role-playing class. ♦

Other possible uses of the *Infer Type* refactoring are the application of the Dependency Injection pattern [Fowler 2004] and the Interface Segregation Principle [Martin 1996]. See also [Infer Type] for more.

3 POSSIBLE, DECLARED, AND INFERRED SUPERTYPES

Every statically typed object-oriented program comes with a set of types, **T**. Every type $T \in \mathbf{T}$ declares a set of members, $\mu(T)$, the subset

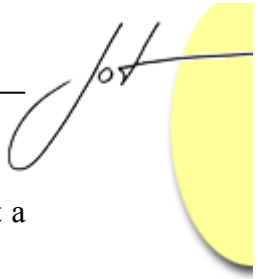
$$\pi(T) := \{m \in \mu(T) \mid m \text{ is a public nonstatic method}\}$$

of which we call the *protocol of the type*. Our focus on protocol — rather than all members of a type — is justified in Section 8.

The subset relation on the protocols of the types in **T** induces a reflexive and transitive relation, $\leq$, on **T**. We write $A \leq B$ for $\pi(A) \supseteq \pi(B)$ with $A, B \in \mathbf{T}$ and call A a *possible subtype* of B. As usual, we extend the notation and write $A \sim B$ for $A \leq B \wedge B \leq A$, as well as $A < B$ for $A \leq B$ and not $A \sim B$. Note that because $A \sim B$ (meaning that $\pi(A) = \pi(B)$) does not imply the identity of A and B, $\leq$ is not antisymmetric and $\leq$ does not define a partial ordering on types. $\leq$ is also known as *structural type conformance*, in contrast to *type conformance by name*, which we consider next.

A program may choose to declare some of the possible subtype relationships contained in $\leq$ as actual (programmatic) subtype relationships. We write `A :<B` to express such declarations (`A implements B` or `A extends B` in JAVA syntax) and call `A` a *declared subtype* of `B`. Note that unlike $\leq$, `:<` is not induced by protocol inclusion, but must be explicitly stated (which is why it is also called *subtyping by name*). Also, `:<` is not reflexive, and `A:<A` is illegal. That `:<` is not at conflict with $\leq$, i.e., that for all $A, B \in \mathbf{T}$: `A:<B` $\Rightarrow A \leq B$, is guaranteed by the programming language's rules of inheritance, by which the protocol of a type is inherited to all its subtypes. Note that it is possible that different types, even if unrelated by subtyping, have identical protocol. We write $A \equiv B$ to denote that (the metasyntactical variables) A and B denote the same type of **T**.

In statically typed languages, types are used to give all expressions of a program a declared (static) type. Expressions comprise user-defined variables (including instance variables, formal parameters, and temporaries), the special variables `this` and `super`, methods with non-void return types (functions in standard programming language terminology), `new` statements (instantiations), and cast expressions. In the following we refer to variables (excluding `this` and `super`) and non-void methods collectively as *references* ("declaration elements" in [Tip et al. 2003]). References can be assigned objects,



which is why they have a *dynamic type*, which is the type of the object they refer to at a certain point in runtime.

Besides its declared type, every expression also has an *inferred type*. For our purposes, we define the inferred type of a reference `a` as that type `I` whose set of members, μ(`I`), is the smallest set containing all members accessed on `a`, united with the sets accessed on all references `a` gets assigned to, both directly and indirectly. Technically, we define μ(`I`) as a function of `a`, ι(`a`), which can be computed by analysing the "forward flow" of (the contents of) `a`, which is specified as the transitive closure of the assignments starting with `a` on the right-hand side. Since we target at statically typed languages, *Static Class Hierarchy Analysis* as described in [Dean et al. 1995] suffices for computing ι(`a`) (*cf*. Section 9).

Generally, the idea of type inferencing is that if all expressions of a program are declared with their inferred type, the program is (still) type correct and there is no other program whose types are more general, i.e., contain fewer members. Type inference is usually used to compute type annotations for untyped (or partially typed) languages [Palsberg & Schwartzbach 1991], but it can also be used to verify problematic constructs in a program (e.g., downcasts [Wang & Smith 2001]), or as the basis of refactoring the type hierarchy (see Section 9).

The inferred type of a reference can contain fields and non-public or static methods, in which case decoupling through a (JAVA or C# style) interface is impossible. Since we may assume that it is also undesired (*cf*. Section 8), we drop all such cases from our further considerations and look only at references `A a` for which ι(`a`) ⊆ π(`A`).

The inferred type of a reference may coincide with its declared type or some other type in **T**, but generally, this will not be the case. However, we know by the rules of static typing that the protocol π(`I`) of the inferred type `I` of a reference declared as `A a` must always be a subset of the protocol of its declared type, π(`A`). Thus, if the inferred type is in **T**, then it is at least a possible supertype of the declared type; if a corresponding subtype declaration exists, it is even a declared supertype. If the inferred type does not exist in **T**, it can be introduced.

In order to maintain type correctness, the introduction of an individual inferred type to a program (together with the introduction of necessary subtype declarations) must follow certain rules, which will be dealt with in Section 5. As we will argue next, the inferred type of a reference — together with its declared type — also provides a measure of the unnecessary amount of coupling established by that reference (its potentially needless "choosiness"), indicating a "bad smell" [Fowler 1999] suggesting a refactoring to decrease that coupling.

4 MEASURING DECOUPLING

Typed references express unilateral coupling between types, because the definition of the type holding a reference depends on the definition of the reference's type. Coupling of a

type with others can therefore be quantified as the number of imported types, for instance by the *Coupling Between Objects* (CBO) metric [Chidamber & Kemerer 1994]. However, in presence of subtyping different imported types express different degrees of coupling: at one extreme, if an imported type is declared final, the coupling is maximal, since only instances of this type are allowed; at the other extreme, if the imported type is the root of the type hierarchy (e.g., `Object`), the coupling is minimal, because any instance is allowed. Clearly, in all but a few pathological cases it is not possible to minimize coupling simply by replacing the references' types with `Object`: static typing rules enforce that the type of a reference must offer at least the protocol invoked on that reference. But if the type offers methods in excess of what is actually needed by the reference, using the type is overly specific in the given context, and resulting coupling is unnecessarily strong.

Intuitively, it would seem that for a declaration `A a`

$$\frac{|\iota(a)|}{|\pi(A)|},$$

the quotient of the number of methods needed from a reference `a`, and the number of methods provided by its declared type `A`, indicates the amount of unnecessary (or excessive) coupling established by `a`: a degree of 1 expresses that all features are needed so that coupling cannot be reduced, whereas one of 0 implies that none of `A`'s features are used, so that the coupling is maximally unnecessary. Its difference from 1 is therefore a measure of the possible reduction of the coupling established by the reference's type; we call this measure the *Actual Context Distance* (ACD) of the reference and its type. For a declaration `A a` it is defined by

$$\mathrm{ACD}(a, A) := \frac{|\pi(A)| - |\iota(a)|}{|\pi(A)|}.$$

For instance, ACD(*c*, *Client*) in the original version of Example 2 of Section 2 is 1/2, and ACD(*s*, *Server*) is 1/3.

Now the actual context distance of a reference and its type can be reduced by replacing its declared type with a more general one, i.e., one that has less excessive protocol. Since such a type can sometimes be found among the declared supertypes of the current type, contemporary IDEs such as ECLIPSE and INTELLIJ IDEA come with special refactorings offering the replacement (and with it a reduction of the ACD) where (if) possible. However, such a type — if it exists — may still hold excessive features, so that replacing the present type with its best available generalization may still leave a positive context distance. We call the context distance in whose computation the reference's declared type has been replaced by the most general of its useable declared supertypes the *Best Context Distance* (BCD) of a reference. (BCD values cannot be computed for the above example, because there are no such supertypes available.) BCD is a measure of the least coupling that can be achieved using existing types only, and ACD – BCD is a measure for the improvement of decoupling possible simply by changing the reference declaration. Thus *introducing* new supertypes for a type can improve (i.e., decrease) *B*CD values, whereas *using* these (or other) abstractions improves *A*CD. An ACD value of 0 is the best possible

achievable; for practical reasons, however, it cannot always be reached (Section 8). Yet, in the refactored example ACD values of both `c` and `s` drop to 0.

In order to be able to assess the actual context distance and the degree to which it can be improved globally, we have extended the metrics from individual references to types. The actual context distance of a type *A* is then defined as the average of ACD(*a*, *A*) over all of *A*'s references, i.e., as

$$\mathrm{ACD}(A) := \left(1 / \sum_{A\,a} 1\right) \cdot \sum_{A\,a} \frac{\left|\pi(A)\right| - \left|\iota(a)\right|}{\left|\pi(A)\right|}$$

and the best context distance (BCD) is defined accordingly. Introducing these global metrics makes sense since a newly introduced interface may be usable in other places, thus altering other references' (of the same or some declared subtype) BCD values. In fact, since available refactorings such as *Use Supertype Where Possible* affect the whole program rather than selected references, using these generalized measures we could even judge the impact of performing such a refactoring on program-wide decoupling.

5 REFACTORING FOR INTERFACE-BASED PROGRAMMING

As outlined in the previous section, the decoupling of a design can be improved by

a) *using* existing, better suited (more general) interfaces, or by
b) *introducing* new interfaces tailored to suit a certain context, and using them.

Refactorings for a) exist; in ECLIPSE, these are named *Use Supertype Where Possible* and *Generalize Type*. However, refactoring for b), the introduction of new interfaces — context-specific ones especially — is left to the wisdom of the programmer: although support for copying method declarations of a class into a new interface is offered, the selection process (i.e., the choice which methods to include) burdens the user (*cf.* discussion of related work in Section 9).

As suggested by the motivating examples in Section 2, our goal is to introduce new interfaces one at a time, starting from an individual reference and its use in a given context. For this purpose, we have to compute the function ι as defined above, i.e., compute the type containing all and only the protocol needed from the chosen reference and all other references it can get assigned to. As argued in Section 3, this type, if not already existent, can be introduced to the program as a supertype of the reference's declared type, replacing the latter in the declaration. However, this is only seldom sufficient, as the following considerations show.

Assume that the declaration is `A a`, and that `I` is the new (maximally general) type computed by ι. If `I` ~ `A` (meaning that both have identical protocol; *cf.* Section 3), `A` is also maximally general and no changes to the program are necessary. If `A` < `I` and `a` terminates an assignment chain (i.e., if it does not get assigned to other references), adding `A:<I` is all that needs to be done: it leaves all assignments to `a` — now declared as `I a` — type correct, and the protocol of `A` remains untouched. If however `a` gets assigned to other

references, these may be no longer declared with the same type as, or supertypes of, `a`'s declared type (which is now `I`), the program thus being type incorrect. Further changes to the program may therefore become necessary, including the possible change of the declared types of other references. However, as will be seen, these are all covered by what we already have at hand, namely the computation of ι on program references.

We deduce the sufficiency of ι from the examination of the following primitive scenario. Let there be two declarations `A a` and `B b`, an assignment `b=a`, and let `I` be the inferred type of `a` (with $\pi(I) = \iota(a)$). Since we start with a (statically) type correct program, we know that:

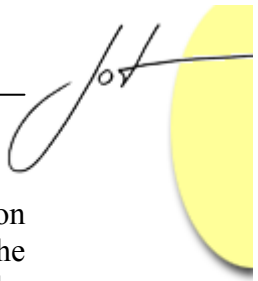
1. $A \equiv B$ or `A:<B` by the typing rules of the programming language.
2. $A \leq I$ by construction of `I`.
3. $\pi(I) = \iota(a) = \iota(a) \cup \iota(b) \supseteq \iota(b)$ by construction of `I`, i.e., $\pi(I)$ contains all and only the methods invoked on `a` or `b`, or the variables they get assigned to.
4. $\pi(B) \supseteq \iota(b)$, i.e., $\pi(B)$ contains all the methods invoked on `b`. If it contains additionally
   a) some or all of the methods invoked on `a`, but no other (so that $\pi(B) \subseteq \iota(a)$), then $I \leq B$.
   b) all of the methods invoked on `a`, plus some invoked neither on `a` nor `b` (so that $\pi(B) \supset \iota(a)$), then $B < I$.
   c) some methods invoked neither on `a` nor `b`, but lacks some methods invoked on `a` (so that $\pi(B) \setminus \iota(a) \neq \emptyset$ and $\iota(a) \setminus \pi(B) \neq \emptyset$), then neither $B \leq I$ nor $I \leq B$.

Because of fact 2, we can declare `A:<I` and re-declare `a` to be of type `I` (i.e., `I a`). As above, all assignments to `a` that were previously type correct still are, and the protocol of `A` remains unaffected. Since `b=a` must also be type correct and the declared type of `a` is now `I`, we must

- either declare `I:<B` or, if this would add unwanted methods to `I`,
- replace `B` in the declaration of `b` by some `J` (possibly `I`) such that $I \leq J$, make sure (in case not $I \equiv J$) that `I:<J`, and declare `B:<J` (in order to keep other assignments to `b` type correct).

Based on the above listed facts, the following four cases must be distinguished (note that unlike for `a`, the new type for `b` need not be maximally general):

i. $A \equiv B$. In this case, `B` can be replaced by `I` in `b`'s declaration, since `I` includes the protocol invoked on `b` (fact 3 above).
ii. `A:<B` and $B < I$. In this case, we can let `B:<I` and replace `B` with `I` in the `b`'s declaration as above.
iii. `A:<B` and $I \leq B$. In this case, declaring `I:<B` is all that needs to be done; in particular, the type of `b` can remain unchanged. However, JAVA's type system sometimes prevents this refactoring, because interfaces cannot subtype classes, and classes can subtype at most one other class directly. More on this in Section 8.
iv. `A:<B` and neither $B \leq I$ nor $I \leq B$. In this case, `B` contains methods that are not in `I` (which are superfluous, since `I` regards all uses of `b`), and `I` contains methods that are not in `B` (which are invoked on `a`, but not on `b` or any other references fur-



ther down in the assignment graph). In this case, `b` must be typed with a common supertype of `B` and `I`, `J`, whose protocol can be obtained as the intersection of the protocol of `B` and `I`, or by applying ι on `b`. In both cases, `B:<J` and `I:<J` must be added to satisfy the typing rules; however, these declarations are guaranteed to leave `A`, `B`, and `I` unaltered, since $A \leq B \leq J$ and $I \leq J$.

Thus we have shown that computation of ι is all that is needed to maintain a type correct program: if a reference is not assigned to any other, replacing its declared type by its inferred type and letting the declared type subtype the inferred type suffices. If it is assigned to other references, their declared types have to be changed and subtype declarations have to be added as described above. New types, if necessary, can be derived by repeated computation of ι.

Two things deserve mention:

- In case of $I \sim B$ (covered by case iii above), we do not replace the new type `I` with the existing type `B`, since this would propagate `B` to other assignments `a` is involved in, potentially letting `B` become a declared subtype of types it was not intended to be. In fact, by obeying the above rules we make sure that new types are inserted in the supertype chain of the original reference's declared type, avoiding inadvertent subtyping (*cf.* Section 9).
- If `b` is a formal parameter of an overridden method, `b=a` is actually a set of assignments, one for each overriding. Whenever a new type is inferred for one parameter declaration `B b`, all others must be changed accordingly, or overriding becomes overloading. A potential problem of changing parameter types is that previously unambiguous method calls may become ambiguous; this is discussed in Section 8.

A question that arises from this description is how costly replacing a reference's declared type is, i.e., how many declarations have to be changed and how many new types the use of a single inferred type induces. As it turns out, in cases i and ii the new type `I` is propagated to subsequent assignments `c=b`, where the procedure must be applied recursively (on `c`). Case iii terminates the ripple effect of changes to a program imposed by the introduction of an inferred type; it necessarily occurs if type `B` is maximally general, i.e., if $\pi(B) = \iota(b)$. Only in case iv, using the inferred type in a reference's declaration can lead to the creation of further new types. Although this could cause an undesirable inflation in the number of types needed to maintain type correctness, we have found (by automatically applying the procedure to all references of several programs) that these cases are rather rare. In fact, it seems that most assignments (including all circular ones) fall under case i, closely followed by case iii. Recursive occurrence of case iv (which is by far the rarest) is limited by the depth of the type hierarchy, since type `C` of `c` would have to be a supertype of both `A` and `B`.

In Section 7, we will present some concrete numbers of the cost of rigorously using inferred types in real projects.

6 THE *INFER TYPE* REFACTORING FOR ECLIPSE

We have implemented the described algorithm as an ECLIPSE refactoring named *Infer Type*, which derives the new type(s) from a selected reference, automatically inserts it/them in the type hierarchy and redeclares all affected references. The refactoring covers the complete JAVA 2 language specification (including arrays, inner types, and anonymous classes) and has been tested extensively by automatically applying it to all references in several large program bases. It can be obtained from [Infer Type].

Applied to the example from Figure 1, the *Infer Type* refactoring works as follows. First the user selects a reference declaration in an editor and chooses *Infer Type* from the context menu. *Infer Type* then computes the new interface by performing an in-depth analysis of the use of the variable in the given program, and presents its result to the user. If one or more interfaces that suit the variable already exist, *Infer Type* presents these for selection; otherwise, it presents the new interface to be created. The user can then enter a name, and *Infer Type* will proceed inserting the new interface in the type hierarchy, redeclaring the variable accordingly, and recompiling the project. The whole process is depicted in Figure 2.

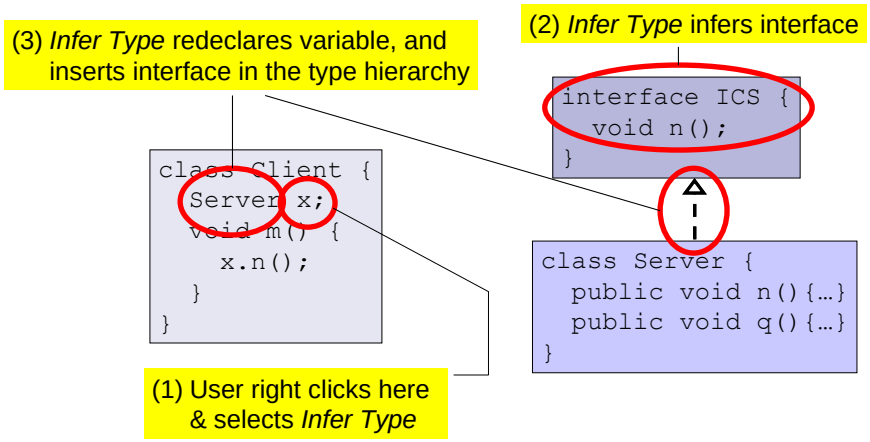
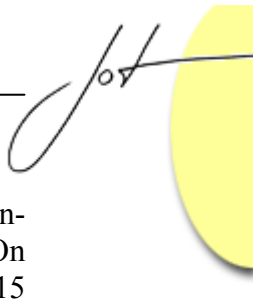


Figure 2: The three steps involved in the refactoring.

7 THE COST OF INTRODUCING INFERRED INTERFACES

While by now it should be clear what the benefits of interface-based programming and our *Infer Type* refactoring are, the costs incurred by the consistent use of context-specific (inferred) interfaces in reference declarations have not yet been addressed. To measure these costs, we have counted the number of new types (interfaces) in programs in which *Infer Type* has been automatically applied to all references contained, as well as the number of variables with (new) types.

The following table shows the results for two medium size projects, DRAWSWF (Version 1.2.9) and JHOTDRAW (Version 6.0 beta 1). Both projects are similar with re-



spect to application domain (they deal with graphics) and size (on the order of a few hundred types). Also, the class-to-interface ratio is roughly the same (approx. 4:1 vs. 3:1). On the other hand, the projects differ greatly with regard to interface utilization: it is about 15 times higher for JHOTDRAW than for DRAWSWF. This may be indicative of the fact that JHOTDRAW is a framework, whereas DRAWSWF is an application using several other frameworks (including AWT and SWING), and also perhaps of different programming styles: JHOTDRAW (which was initially developed by Erich Gamma) is programmed against interfaces, whereas DRAWSWF is not. This observation is corroborated by the measured average ACD values: they are comparatively high for DRAWSWF (0.416 on average), suggesting that coupling is unnecessarily strong, and low for JHOTDRAW (0.142). Note that the gains possible by using better suited (more general) existing types are moderate in both cases, as indicated by the very small differences between ACD and BCD values: they suggest that programmers used the supertypes that were available.

	DRAWSWF			JHOTDRAW		
	before	after	change	before	after	change
Types	346 (100%)	599 (100%)	+253 (+73%)	395 (100%)	717 (100%)	+322 (+82%)
Classes	282 (82%)	282 (47%)	(±0%)	301 (76%)	301 (42%)	(±0%)
Interfaces	64 (18%)	317 (63%)	+253 (+395%)	94 (24%)	416 (58%)	+312 (+343%)
Ratio	1:0.23	1:1.12	× 1:5.0	1:0.31	1:1.38	× 1:4.5
References	1361 (100%)	1361 (100%)	(±0%)	1801 (100%)	1801 (100%)	(±0%)
Class typed	1109 (81%)	404 (30%)	–705 (–64%)	395 (22%)	81 (4%)	–314 (–79%)
Interface typed	252 (19%)	957 (70%)	+705 (+280%)	1406 (78%)	1720 (96%)	+314 (+22%)
of these empty		120 (9%)			146 (8%)	
new types only		709 (52%)			1454 (81%)	
Ratio	1:0.23	1:2.37	× 1:10.3	1:3.56	1:21.23	× 1:6.0
Refs. per type	3.93	2.27	× 0.58	4.56	2.51	× 0.55
new types only		2.80			4.52	
Average ACD	0.416	0	(–100%)	0.142	0	(–100%)
Average BCD	0.351	0	(–100%)	0.100	0	(–100%)
Av. ACD–BCD	0.065	0	(–100%)	0.041	0	(–100%)

To some surprise, the changes induced by applying *Infer Type* to all references do not differ significantly between the projects: the total number of types increases by 73% and 82%, respectively, with the number of interfaces more than tripling in both (395% and 343%). Note that while the number of class typed references in JHOTDRAW fell to 81 (4%), that of DRAWSWF remained at a comparatively high level (404 or 30% of all references). This is due to the fact that DRAWSWF, unlike JHOTDRAW, makes heavy use of frameworks that are not part of the project, and that *Infer Type* does not attempt to refactor the type hierarchy of external code. The relatively moderate number of new types in both projects can be explained by a notable degree of reuse: overall, each new interface can be used for 2.8 reference declarations in DRAWSWF and for 4.5 in JHOTDRAW.

Despite the less than what one might expect increase in the number of types, replacing all types in a program with their inferred types is clearly impractical. For once and as stated above, it is not useful to derive context-specific interfaces for all variables; instead, one would focus on the plug point variables of a system, of which there are usually only few. On the other hand, a closer analysis of the derived data shows that the newly introduced interfaces are of largely differing popularity: as indicated in Figure 3, as few as 31 and 26 new interfaces account for as many as 50% of the retyped variables in DRAWSWF and JHOTDRAW, respectively. In fact, the number of variables typed with each new interface roughly follows a Pareto distribution; presumably, fewer still can account for even more variables, if interfaces of a class that differ in only few methods are merged (a kind of interface clustering; see Section 10). A more detailed discussion of these and other numbers can be found in [Forster 2006].

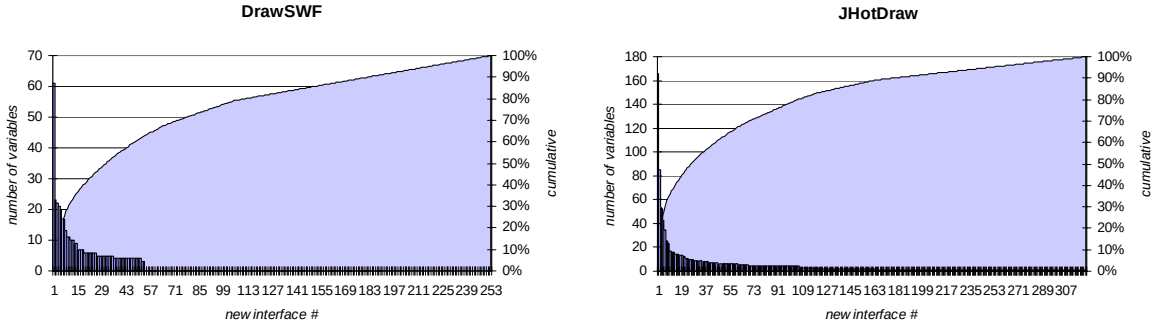


Figure 3: Number of references per new interface.

Concerns are sometimes raised that method invocation through interfaces implies a penalty on runtime efficiency. Although this explains a certain reservation of programmers against the use of interfaces in variable declarations, more recent improvements in the implementation of `invokeinterface` suggest that this fear is no longer justified (see, e.g., [Alpern et al. 2001]).

8 DISCUSSION

Language particularities

The application of metrics and refactorings to a specific programming language has to take the language's peculiarities into account. We chose JAVA as our target language, mainly because our work is practically oriented and JAVA is widely used. However, C#'s great similarities with JAVA should make adaptation of our work a fairly easy exercise. In fact, C#'s type system (which is very much in line with Microsoft's general policy to favour interfaces and delegation over inheritance) allows one to declare features of a class as accessible only via an interface [Microsoft C#]. Adaptation of our work to other languages (in particular those that make no syntactic distinction between an interface and a class) is of course possible, as long as multiple (interface) inheritance is allowed.

Non-publicity and decoupling. In JAVA, access to public attributes (fields) and non-public members may prohibit the use of an interface. As for the former: direct access to public fields can be replaced by accessor methods, which can then be added to the interface. This is usually considered good practice, supported — even without changing the caller's syntax — by the concept of *properties* or *features* in languages such as C#, DELPHI and EIFFEL, and enforced by languages with stronger notions of information hiding such as SMALLTALK (where instance variables cannot be accessed directly even from other objects of the same class). As for the latter: decoupling from private members is usually not an issue, nor is that of protected ones (since inheritance establishes a stronger coupling between classes than the referencing through variables or methods). Default (package local) members (which are also excluded from interfaces) support JAVA's package concept, which is meant to present some kind of modularization on top of the class level. Since coupling within a module is usually a goal rather than a flaw, we assume that abstraction from package local type access through a decoupling interface is not what is wanted, allowing one to regard all package local features as if they were declared private.

Occasional problems with supertyping. When case iii of Section 5 occurs, the inferred type must be a subtype of an existing type, which can be a class. Unless this class is `Object`, insertion of an interface is prohibited by JAVA's type system. In this case, our current implementation of *Infer Type* tries to insert an abstract class. However, this possibility is limited by the fact that in JAVA, classes can have only one direct superclass. Therefore, if an occurrence of case iv requires that `I` must be introduced as a sibling to `B` (with `A` being a subtype of both) and the common supertype of `I` and `B`, `J`, must be a class (because `b` gets assigned to `c` and `c`'s type is a class; see above), *Infer Type* gives up. Alternatively, it could replace `c`'s type with an interface, but this change would have to propagate right to the terminal nodes of the assignment graph, which is presumably not what is desired.

For the problem of letting a library type subtype a (new) user-defined type ("retroactive type abstraction") we refer the reader to the treatment in [Läufer et al. 2000].

Problems of metrics

The domain of metrics is "deep and muddy waters" [Henderson-Sellers 1996]. Our metrics introduced here serve mainly two purposes: they aid in deciding whether to introduce a new interface or use an existing one in a certain context, and they measure the effect global refactorings such as *Use Supertype Where Possible* have on the decoupling of a program. Other metrics of interface-based programming we developed have successfully been employed to detect so-called enabling interfaces in code [Steimann & Mayer 2005]; however, all usual objections against metrics apply. In particular, our metrics provide no objective assessment of the quality of the code; at best, different programming styles can be identified.

Problems of refactorings

Effects on program semantics. Inferred types of a correctly typed program are always supertypes of the declared types. Replacing a reference's declared type with an inferred type therefore leaves all assignments to that reference type correct. In fact, one might believe that the behaviour of an otherwise untouched program remains unchanged if inferred types consistently replace the declared ones. However, the affected references now accept more values (objects of different types) than before (which is why refactorings in the sake of decoupling are performed in the first place). This makes not only uses of parts of the program exposing the references, by others (as well as changes to the program itself) possible that were impossible before the refactoring, it also allows more input from the user, for instance if the user interface of an application offers a list of objects to pick from, where the list depends on the type of the input variable (as is the case in a drag&drop scenario). Thus, we maintain that the semantics of a program changes due to the refactoring (which is actually the case for most refactorings described in [Fowler 1999]), but suggest that this change is intentional. However, there are other, rather subtle problems.

Ignored implicit contracts. Since *Infer Type* is based on a purely syntactical analysis, it cannot respect the intent of the programmer, even if it is expressed in the code. The following counterexample illustrates this.

Revisiting Example 1, if `Set` (or even `HashSet`) rather than `Collection` were used as the formal parameter type in the interface of the component, replacing it with `Iterable` would break the implicit contract of the server with the client, namely the latter's obligation to pass no collections containing duplicates. This is so because interfaces in JAVA are purely syntactical constructs, and even if the type inference algorithm would detect that there is such a restriction in the implementation of `HashSet` (and other types realizing the interface), it would have no means of equipping the abstraction `Iterable` with a corresponding constraint, simply because this is not supported by the language.

Overloading and ambiguous method calls. Due to its single dispatch policy, JAVA must distinguish between overloading and overriding. Overloading of methods entails the danger of introducing ambiguous method calls, i.e., method call sites that cannot be statically bound to a unique strain of (overridden) method implementations. This is for instance the case if two overloaded variants of a method exist whose formal parameter types are both supertypes (without one subtyping the other) of the declared type of the actual parameter at a call site. Possible ambiguities of this kind are caught by the JAVA compiler.

Now the introduction of a generalized formal parameter via *Infer Type* (or some other refactoring) can make a previously unambiguous method call ambiguous. For example, let `A` of a method signature `m(A a)`, for which an overloaded version `m(B a)` such that `A:<B` exists, be replaced with an inferred type (generalization) `I` such that neither `B` is a subtype of `I` nor vice versa. A call site of `m` with declared (and unchanged) actual parameter type `A` is now ambiguous. Although it is well-known that overloading is a general problem for refactorings [Tip et al. 2003], it is unclear (at least to us) how to avoid it. Ac-

cidental overriding (induced by the refactoring of an overloaded method) is another problem that falls into this category. It is interesting, though, that during our extensive experiments we have not experienced a single occasion of method call ambiguity (but note that accidental overriding cannot be detected by a compiler).

Globalization of locally introduced interfaces. Despite the theoretical bounds mentioned in Section 5, propagation of type replacements (inferred types) through a program may pose a problem if the introduced type is meaningful in the context were it was introduced, but is not where it propagates to (as evidenced, e.g., by the inappropriateness of its name in the new context). Such is particularly the case if a new formal parameter type is introduced to reflect the parameter's role in a collaboration, and passing the actual parameter to another collaboration would make it adopt a new role, rather than taking the old one with it (note that the variable's name changes with every assignment, whereas its type does not). In these cases type casts making the role change explicit (and confining the dissemination of the new type) seem conceptually justified. However, this is another issue. (Note that the propagation can always be terminated by inserting a downcast to the original type, which is safe if the program remains unaltered otherwise. However, this would somehow void the effect of introducing the inferred type, questioning the whole approach. On the other hand, a cast to the next inferred type would require the two to have a common subtype. Conditions under which such casts are always safe still need investigation, but it seems that they are related to issues of role modelling [Steimann 2001].)

Reflection. Considering that reflection can involve computed (and thus unpredictable) names, we have made no attempt at treating reflection.

9 RELATED WORK

Metrics

Extensive experience with metrics has shown that generally valid measures are hard to establish [Henderson-Sellers 1996], and that instead metrics must be tied to a specific goal (the GQM approach [Basili et al. 1994]). The goals of the metrics presented in Section 4 are clear cut: to measure the use of interfaces in interface-based programming. Since no metrics for this purpose existed, the definition of our own suite seemed justified.

Refactorings and type inference

Several algorithms for type inference have been described in the literature. Some are based on solving a set of type constraints attached to the declaration elements of a program (e.g. [Palsberg & Schwartzbach 1991, Tip et al. 2003, Wang & Smith 2001]), while others rely on a data flow analysis of the program (see [Khedker et al. 2003] for an overview and in-depth comparison with constraint-based type inferencing). Data flow analysis as well as the generation of constraints can be based on a static or a dynamic analysis of a program's call graph; although the latter is more precise (the resultant types are more

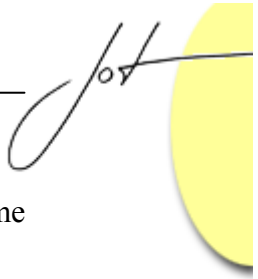
general, i.e., have fewer elements), the static typing rules of languages such as JAVA prevent them from being used in a program. Therefore, static analysis is perfectly accurate for our purposes, which frees us from theoretical issues such as the tractability of precise dynamic flow analysis.

Type inference algorithms for JAVA have been implemented as parts of IDEs such as ECLIPSE and INTELLIJ IDEA. ECLIPSE's implementation is based on constraint satisfaction and is used in the refactorings *Use Supertype Where Possible* and *Generalize Type* [Tip et al. 2003]. Both refactorings rely on the availability of suitable interfaces rather than providing them: they check — rather than compute — possible solutions. Implementations of *Extract Interface* in ECLIPSE and INTELLIJ IDEA require programmers to design their interfaces manually; a newly defined interface can then be used in variable declarations throughout the program, where possible. This is in contrast to our approach, in which we compute a new interface, constructed to be used mainly for the reference from whose context it was derived.

Snelting & Tip have presented an algorithm based on formal concept analysis that computes for a given program a new type hierarchy containing all maximally general (inferred) types [Snelting & Tip 2000]. Its effect appears to be roughly the same as that of global type inference in the spirit of [Palsberg & Schwartzbach 1991]. Streckenbach & Snelting have applied this algorithm to the refactoring of JAVA programs, automatically changing all type references (including instantiations) in a program (the KABA system) [Streckenbach & Snelting 2004]. KABA also offers a refactoring tool for collapsing and manually reorganizing the produced type hierarchy, but this operates on (compiled) byte code of an application. It is unclear to us if their algorithm would also work on individual references (rather than all expressions of a package); in any case, it has not been integrated into an IDE, which is in accord with the fact that operating on byte code gives it the status of a post processor in the spirit of JAX [Tip et al. 2002].

One general goal of refactoring class hierarchies is to maximize the (functional) cohesion of modules, i.e., to ensure that all members of a class are always used together, thus minimizing (the footprint of) objects (e.g., [Opdyke & Johnson 1993, Streckenbach & Snelting 2004]). For this, new subclasses must be introduced, which is why the work is also referred to as *program* [Streckenbach & Snelting 2004] or *class hierarchy specialization* [Tip & Sweeney 2000]. KABA uses points-to analysis to identify specialization candidates; due to its approximating nature, this introduces certain artifacts [Streckenbach & Snelting 2004]. By contrast, we are interested only in *program generalization*, opening it up for wider reuse, increasing decoupling. Consequently, we do not introduce new concrete classes, nor do we touch instance creation; points-to analysis is therefore also not needed.

Other work on type inference in statically typed object-oriented programming languages aims at making downcasts safe without guarding them [Wang & Smith 2001]. This requires a proof that all objects an expression being downcast evaluates to are of the target type of the cast or some subtype thereof. This problem is somewhat converse to ours, not only because we are interested in type generalizations (specifically: the maxi-



mum allowable upcast), but also since it requires an analysis of where the objects come from rather than where they go.

An often overlooked problem with automated refactorings of type hierarchies based on structural conformance (including those based on pure concept analysis) is that they cannot deal with accidental conformance, potentially compromising the (intended) semantics of a program. For instance, covariant redefinition of an instance variable (field) may introduce different referenced subtypes with identical protocol. Now an automated refactoring of the type hierarchy might be tempted to merge these into one. However, the variables were redefined covariantly in order to keep their value domains separate — joining the types would breach the intended semantics of the program, since the variables can now be assigned equal — even identical — values. Inadvertent merges of unrelated or intentionally separated types are particularly disastrous if they introduce multiple (interface) inheritance, because this propagates a cross-over assignment compatibility of formerly disjoint branches of the type hierarchy to all subtypes.

Generally, one of the biggest problems with the automatic generation of types (and unguided refactoring of type hierarchies) is its inability to find suitable names for the new types. Indeed, much of the intended semantics of a program is captured in the type names, which is reflected in the fact that finding (reasonably) good names is usually no problem for the author of the program, as opposed to finding good names for a (decompiled or scrambled) program when reading it (if one is not the author). Programmers are likely to perceive a complete refactoring of their type structure (even though it may be eye-opening in certain cases) as dull, simply because it ignores their intentions. We respect this author (or director) role of the programmer and offer the opportunity to compute a new type where deemed appropriate, or where suggested by a metric.

10 NEXT STEPS

The introduction of parametric polymorphism (generics) to JAVA 1.5 hits all tool builders hard. It requires redefinition of all refactorings and metrics relating to the type hierarchy, including our *Infer Type*. Appropriate extensions of our algorithm that addresses subtyping of generic types are currently being developed. Another issue on our agenda is to find natural ways of limiting the ripple effect the introduction of new interfaces may have on variable declaration. A possible solution to this problem has already been sketched in Section 8. The development of another plugin to ECLIPSE that adds warnings and quick fixes for all variable declarations in which an existing or newly derived interface could be used will be completed shortly. Last but not least, we are currently working on tools for interface mining, i.e., on the automatic clustering of similar interfaces with the purpose of maximising the utility of a minimum number of interfaces to be introduced.

11 CONCLUSION

Despite the obvious benefits of interface-based programming, the majority of programmers seem to be slow if not to use existing interfaces, at least to create new ones. We conjecture that with the available tools at hand this can be only partly blamed on the extra coding effort imposed by the separate maintenance of an interface and its implementing classes, or on the fear of compromised runtime efficiency. A major obstacle seems to be the uncertainty regarding where interfaces should be introduced, and what they should contain. With our type inference algorithm, its associated metrics, and our *Infer Type* refactoring, we believe to have provided a highly usable tool that guides and aids the programmer with this important design decision.

ACKNOWLEDGMENTS

The author wishes to thank Philip Mayer und Andreas Meißner for implementing the *Infer Type* refactoring including its associated metrics, and Florian Forster for continuing their work and delivering the numbers presented in this article.

REFERENCES

[Alpern et al. 2001] B Alpern et al. "Efficient implementation of Java interfaces: Invokeinterface considered harmless" in: *OOPSLA* (2001) 108–124.

[Basili et al. 1994] VR Basili, G Caldiera, D Rombach. "The goal question metric approach" in: *Encyclopedia of Software Engineering* (John Wiley & Sons, 1994).

[Canning et al. 1989] PS Canning, WR Cook, WL Hill, WG Olthoff "Interfaces for strongly-typed object-oriented programming" in: *Proc. of OOPSLA* (1989) 457–467.

[Chidamber & Kemerer 1994] SR Chidamber, CF Kemerer "A metrics suite for object oriented design" *IEEE TSE* 20:6 (1994) 476–493.

[Dean et al. 1995] J Dean, D Grove, C Chambers "Optimization of object-oriented programs using static class hierarchy analysis" in: *Proc. of ECOOP* (1995) 77–101.

[Forster 2006] F Forster "Cost and benefit of rigorous decoupling with context-specific interfaces" in: *Proc. of PPPJ* (2006) 23–30.

[Fowler 1999] M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).

[Fowler 2004] M Fowler "Inversion of control containers and the Dependency Injection pattern" *http://www.martinfowler.com/articles/injection.html*.

[Gamma et al. 1995] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns — Elements of Reusable Software* (Addison-Wesley, 1995).

[Gößner et al. 2004] J Gößner, P Mayer, F Steimann "Interface utilization in the JAVA Development Kit" in: *Proc. of SAC 2004* (ACM, 2004) 1310–1315.

[Henderson-Sellers 1996] B Henderson-Sellers *Object-Oriented Metrics: Measures of Complexity* (Prentice Hall 1996).

[Infer Type] Infer Type documentation *http://www.fernuni-hagen.de/ps/prjs/InferType*.

[Khedker et al. 2003] UP Khedker, DM Dhamdhere, A Mycroft "Bidirectional data flow analysis for type inferencing" *Computer Languages, Systems & Structures* 29:1–2 (2003) 15–44.

[Läufer et al. 2000] K Läufer, G Baumgartner, VF Russo "Safe structural conformance for JAVA" *The Computer Journal* 43:6 (2000) 469–481.

[Martin 1996] RC Martin "The Interface Segregation Principle" *http://www.objectmentor.com/resources/articles/isp.pdf* (1996).

[Microsoft C#] Microsoft *C# Language Specification* (http://msdn.microsoft.com/)

[Opdyke & Johnson 1993] WF Opdyke, RE Johnson "Creating abstract superclasses by refactoring" in: *ACM Conf. on Computer Science* (1993) 66–73.

[Palsberg & Schwartzbach 1991] J Palsberg, MI Schwartzbach "Object-oriented type inference" in: *Proc. of OOPSLA* (1991) 146–161.

[Pattison 2000] T Pattison *Programming Distributed Applications with COM & Microsoft Visual Basic* (Microsoft Press, 2000).

[Snelting & Tip 2000] G Snelting, F Tip "Understanding class hierarchies using concept analysis" *ACM TOPLAS* 22:3 (2000) 540–582.

[Steimann 2001] F Steimann "Role = Interface: a merger of concepts" *Journal of Object-Oriented Programming* 14:4 (2001) 23–32.

[Steimann et al. 2003] F Steimann, W Siberski, T Kühne "Towards the systematic use of interfaces in Java programming" in: *Proc. of PPPJ* (ACM, 2003) 13–17.

[Steimann & Mayer 2005] F Steimann, P Mayer "Patterns of interface-based programming" *Journal of Object Technology* 4:5 (2005) 75–94.

[Streckenbach & Snelting 2004] M Streckenbach, G Snelting "Refactoring class hierarchies with KABA" in: *Proc. of OOPSLA* (2004) 315–330.

[Tip et al. 2003] F Tip, A Kiezun, D Bäumer "Refactoring for generalization using type constraints" in: *Proc. of OOPSLA* (2003) 13–26.

[Tip & Sweeney 2000] F Tip, PF Sweeney "Class hierarchy specialization" *Acta Informtica* 36:12 (2000) 927–982.

[Tip et al. 2002] F Tip, P F Sweeney, C Laffra, A Eisma, D Streeter "Practical extraction techniques for JAVA" *ACM TOPLAS* 24:6 (2002) 625–666.

[Wang & Smith 2001] T Wang, SF Smith "Precise constraint-based type inference for JAVA" in: *Proc. of ECOOP* (2001) 99–117.

About the author

Friedrich Steimann is a Professor of Informatics at the Fernuniversität in Hagen, Germany. He heads the department of Programming Systems and conducts research on object-oriented programming concepts, software modelling, and development tools. He can be reached as steimann[at]acm.org.