Decoupling Classes with Inferred Interfaces

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
58097 Hagen, Germany
steimann@acm.org

Philip Mayer
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
Universitätsstraße 1
58097 Hagen, Germany
plmayer@acm.org

Andreas Meißner
IBM Ottawa Lab
2670 Queensview Drive
Ottawa, ON K2B 8K1
Canada
meissner@acm.org

ABSTRACT
Using small, context-specific interfaces in variable declarations serves the decoupling of classes and increases a program's flexibility. To minimize its interface, a thorough analysis of the protocol needed from a variable is required. Currently available refactorings for the extraction of interfaces leave the programmer alone with the decision which methods to include or, more problematically, which to omit: they let him choose manually from the protocol of an existing type, and only then offer to use the new interface where (if) possible. To aid the programmer in defining a new interface, we have developed a new refactoring that infers it from a variable's declaration and automatically inserts it into the code.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Classes and objects, Abstract data types, Inheritance.*

General Terms
Design, Languages, Measurement, Theory.

Keywords
Type inference, Interface-based programming, Refactoring.

1. INTRODUCTION
It is common wisdom that object references (variables) should be typed with abstract interfaces rather than concrete classes [4][5]. Ideally, the interfaces would be minimal, i.e., they would contain only the protocol required from the references. This would increase both security and plugability, by restricting access to and by reducing the dependency (as measured in terms of the size of the required protocol) on the referenced objects to what is actually needed rather than what happens to be offered by an available type. However, in practice the protocol required from a reference is often non-obvious, since references get assigned to others, and these assignments force the programmer to look at many difference places in a program (particularly if calls to overridden methods are involved). It is therefore often unclear what the interface should contain, so that in practice, most programmers still prefer to use the original classes in type declarations.

We aim to better this situation by providing a fully automated refactoring that makes the introduction of fitting interfaces an easy exercise. After presenting a motivating example (Section 2) we introduce a few fundamental definitions (Section 3) necessary to understand the formal framework of our type inferring algorithm and refactoring (Sections 5 and 6), as well as the metrics indicating where it should be applied (Section 4). We then discuss our work in light of certain language-specific and other problems (Section 7), and compare it with the work of others (Section 8).

SAC'06, April 23-27, 2006, Dijon, France.

2. A MOTIVATING EXAMPLE
The following piece of JAVA code presents a very simple example of a mutual coupling between a client and a server. The coupling is established via the declarations `Server s` and `Client c`:

```
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() {…}
}
```

Refactored for maximum decoupling, the code should look like

```
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() {…}
}
```

The example is simplistic since what client and server need from each other is obvious; in particular, it need not be extracted by following long assignment chains. However, it suffices to show that with the refactored solution, the server's services can be used by clients that do not possess a `do` method, and clients can use servers unable to `doSomethingCompletelyUnrelated`.

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 7.1.

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 $A = B$ for $A \leq B \wedge B \leq A$, as well as $A < B$ for $A \leq B \wedge A \neq B$. Note that because $A = B$ (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* [9], 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 [18]). 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 [3] suffices for computing ι(`a`) (*cf.* Section 8.2).

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 7.1), we drop all such cases from our further considerations and look only at references `A a` for which ι(`a`) $\subseteq \pi$(`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" [5] 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 [2]. 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`: typing rules enforce that the type of a reference must offer at least the protocol invoked on that reference. But if the type has methods in excess of what is actually needed by the reference, use of the type is overly specific in the given context, and coupling 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 the example 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 "smaller" 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 examples, 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 7.1). Yet, in the refactored example ACD values of both *c* and *s* drop to 0.

5. REFACTORING FOR DECOUPLING

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

a) *using* existing, better suited (less specific) 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) is burdened on the user (*cf.* discussion of related work in Section 8.2).

We have therefore implemented a new refactoring, *Infer Type*, that computes ι as defined above, i.e., that computes the *least specific type containing all protocol needed from the chosen reference and all other references it gets possibly assigned to*. As outlined above, 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 does not generally suffice, as the following considerations show.

The goal is to replace a type `A` in the declaration of a reference `a` by a minimal type `I`, as computed by *Infer Type*. If `I` = `A` (meaning that both have identical protocol; *cf.* above), `A` is also minimal and no changes to the program are necessary. If `a` is not minimal, but 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 not accept the "added values" (more objects) allowed by `a`'s new type `I`, because they are no longer declared with the same type as, or supertypes of, `a`'s declared type (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.

We deduce the sufficiency of our *Infer Type* procedure defined as above 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. $\pi(I) = \iota(a) \cup \iota(b) \supseteq \iota(b)$ by construction of `I`, i.e., $\pi(I)$ contains all and only the methods invoked on `a` or `b`.
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 minimal):

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 7.2.
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 further 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 *Infer Type* 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 $I \leq J$.

Note that in case of `I` = `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. 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 8.2).

Special care must be taken when following the assignments implicit in a method call: if the method is overridden, separate branches of the assignment graph are commenced and the formal parameter type `B` of all overridden definitions must be changed to the same type `I`, or overriding becomes overloading.

Thus we have shown that *Infer Type* 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 *Infer Type*.

Questions that remain are 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 minimal, 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 *Infer Type* to all references of several programs, including JUNIT and JHOTDRAW) that these cases are very 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 practice, we have found that applying *Infer Type* to all references of a program roughly doubles its number of types. A more detailed presentation and analysis of results will be the subject of another paper.

6. AVAILABILITY

We have implemented the described algorithm as an ECLIPSE refactoring that 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 http://www.fernuni-hagen.de/ps/docs/InferType.

7. DISCUSSION

7.1 Decoupling in Java and C#

Non-publicity and decoupling. In JAVA and C#, access to public attributes (fields) and non-public members may prohibit the use of an interface. As for the former: access to public fields can be encapsulated by accessor methods, which can then be added to the interface. This is usually considered good practice, anyway. 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 a package concept, which is meant to present some kind of modularization on top of the type level. Since coupling within a module (called cohesion) 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.

Occasional impossibility of subtyping. The insertion of a new interface below (i.e., letting the interface subtype) a class is prohibited by JAVA's and C#'s type system. Although in such a case an abstract class (rather than an interface) can be introduced, this prevents its subclasses from extending other classes. This may make the introduction of a context-specific type more complicated than described in Section 5. One solution to this problem is to replace the superclass that caused the problem by an interface, for instance by inferring the superclass's type. For the problem of letting a library type subtype a (new) user-defined type ("retroactive type abstraction") see [9].

7.2 Problems with Refactoring

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 could argue 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 uses of those parts of the program exposing the references, by others (as well as changes to the program itself) possible that were impossible before the refactoring. We suggest that this change is intentional.

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 role of a collaboration [14], and passing the actual parameter to another collaboration would make it adopt a new role, rather than take the old one with it (note that the object's reference changes with every assignment, whereas its declared type does not necessarily). 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.

8. RELATED WORK

8.1 Metrics

Extensive experience with metrics has shown that generally valid measures are hard to establish [7], and that instead metrics must be tied to a specific goal (the GQM approach [1]). The goals of the metrics presented in Section 4 are clear cut: to measure the degree of coupling through typed references in object-oriented programs. Since no metrics for this purpose existed, the definition of our own seemed justified.

8.2 Refactoring 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. [12], [21]), while others rely on a data flow analysis of the program (see [8] 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 less specific, i.e., have fewer elements), the static typing rules of languages such as JAVA and C# 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.

Although we did not find an existing type inference algorithm for JAVA, algorithms somewhat related to ours have been implemented as parts of IDEs such as ECLIPSE and INTELLIJ IDEA. ECLIPSE's implementation is based on constraint satisfaction, and used in the refactorings *Use Supertype Where Possible* and *Generalize Type* [18]. However, 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.

Tip & Snelting have presented an algorithm based on formal concept analysis that computes for a given program a new type hierarchy containing all minimal types [13]. Its effect appears to be roughly the same as that of global type inference in the spirit of [12]. Streckenbach & Snelting have only recently applied this algorithm to the refactoring of JAVA programs, automatically changing all type references (including instantiations) in a program (the KABA system) [17]. 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 [20].

An often overlooked problem with automated refactorings of type hierarchies based on structural conformance (including those based on pure concept analysis, but excluding ours, which is based on conformance by name) 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.

One general goal of refactoring class hierarchies (as in, e.g., [11] [17]) is to maximize the (functional) cohesion of a module, i.e., to ensure that all members of a class are always used together, thus minimizing (the footprint of) objects. For this, new subclasses must be introduced, which is why the work is also referred to as *program* [17] or *class hierarchy specialization* [19]. KABA uses points-to analysis to identify specialization candidates; due to its approximating nature, this introduces certain artifacts [17]. 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 (and because our target language is statically type checked; *cf*. above*)* not needed.

Other work on type inference in statically typed object-oriented programming languages aims at making downcasts safe without guarding them [21]. This requires a proof that all objects an expression being downcast can produce (stands for) are at least 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 maximum allowable upcast), but also since it requires an analysis of where the objects come from rather than where they go.

9. CONCLUSION

The interface-as-type construct [16] is sometimes regarded as JAVA's biggest single contribution to mainstream programming. However, the introduction of good interfaces, ones that are specifically designed to decouple a client from its servers especially, remains a tricky problem, since the decision what to put into the interface would require an in depth analysis of the client's and servers' code. To attack this problem, we have presented a theoretical framework for the maximization of decoupling in programs, and cast it into an automatic refactoring that is available as an ECLIPSE plugin. Systematic application of this refactoring has shown that the number of additional types introduced is moderate.

REFERENCES

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

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

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

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

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

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

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

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

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

[10] P Mayer "Analyzing the use of interfaces in large OO projects" *OOPSLA 2003 Companion* (2003) 382–383.

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

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

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

[14] F Steimann "Role = Interface: a merger of concepts" *JOOP* 14:4 (2001) 23–32.

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

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

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

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

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

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

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