JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 5, July-August 2005

Patterns of Interface-Based Programming

Friedrich Steimann, Universität Hannover
Philip Mayer, Universität Hannover

Abstract

Modern software architectures heavily promote the use of interfaces. Originally conceived as a means to separate specification from implementation, popular programming languages toady accommodate interfaces as special kinds of types that can be used – in place of classes – in variable declarations. While it is clear that these interfaces offer polymorphism independent of the inheritance hierarchy, little has been said about the systematic use of interfaces, or how they are actually used in practice. By providing a set of basic patterns of interface use together with numbers of their frequency we provide insights that should be of interest not only to the practising programmer, but also to the designers and analysts of large code bases.

1 INTRODUCTION

After the hype has gone, what remains from a new technology is often not what it has originally been acclaimed for. In the case of object-oriented programming, it seems that the praise for its potential of code reuse through implementation inheritance has vanished in favour of an appreciation of what is sometimes called interface inheritance or, more generally, *interface-based programming* [Pattison00]. This programming style derives from the fact that instances of different classes are allowed to take the same place (i.e., can be assigned to the same variable) as long as they guarantee to conform to the same interface specification, and that this substitutability is independent from whether or not the classes share implementation. In interface-based programming, variables are therefore typed with interfaces rather than classes.

The programming language JAVA has recently popularized the interface-as-type concept: with it, the designers of a program have the choice to code abstract types (types that cannot be instantiated) as either abstract classes or interfaces. The JAVA language specification however is rather reticent about the reason to introduce a separate interface construct; in fact, from reading the documentation one is led to believe that the primary purpose of its existence is to compensate for JAVA's lack of multiple inheritance.

Cite this article as follows: Friedrich Steimann, Philip Mayer: "Patterns of Interface-Based Programming", in *Journal of Object Technology*, vol. 4, no. 5, July-August 2005, pp. 75-94 http://www.jot.fm/issues/issue_2005_07/article1

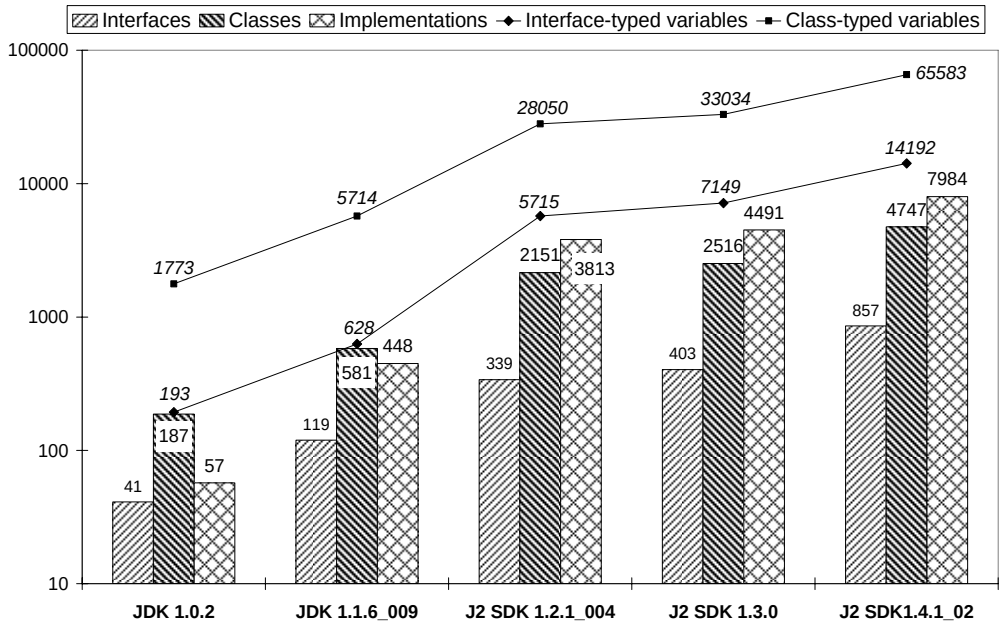
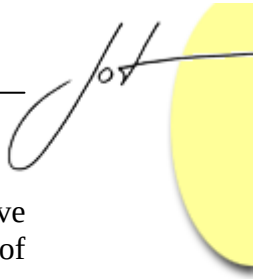


Figure 1. Development of interface utilization in the JDK.

Figure 1 gives an impression of how the use of interfaces has developed over the past five major releases of the JAVA application programming interface (API, called JDK hereafter). Whereas the ratio of classes to interfaces almost remained constant (approx. 5.5:1 across all releases; note that equal ratios translate to equal distances on a logarithmic scale), that of interface-typed to class-typed variables nearly doubled with the change to JAVA 2 (JDK 1.2; approx. from 1:9 to 1:5). At the same time, the average number of interfaces implemented per class jumped from 0.8 to 1.8.

Considering that in JAVA classes must explicitly declare to implement interfaces for their instances to be assignable to correspondingly typed variables, one might assume that the increased availability of interface-implementing classes accounts for the increased popularity of interface-typed (in relationship to class-typed) variables. However, the even more dramatic increase of interface implementations one release earlier (which was more than 2.5-fold) was not accompanied by a corresponding increase in the ratio of interface to class-typed variables (which instead remained almost constant); in fact, it goes back to a large part to the introduction of the `Serializable` interface, which is implemented by 43% of all classes of the JDK 1.1.6, but which has only few (30) variables declaring it as their type. As it turns out, one contributor to the considerable increase in interface-typed variables from JDK 1.1 to JDK 1.2 is the introduction of the JAVA collections framework, whose 11 interfaces account for 13% of all interface-typed variables in 1.2, but – contrary to `Serializable` – come with only few implementing classes. In the same vein, but much more than collections the GUI framework SWING (also introduced with JAVA 2) has contributed to the statistics: its interfaces come with 54% of all interface-typed variables of the JDK 1.2. While this may be taken as a first evidence of the existence of different usage patterns for interfaces, we note here that since then, both the average number of



interface implementations and the ratio of interface to class-typed variables have remained almost constant, suggesting that the use of interfaces, as well as perception of their utility, – at least by the authors of the JDK – has settled.

Outside the realm of JAVA, one can observe that especially within the context of large APIs and frameworks such as CORBA, J2EE, COM or .NET the use of interfaces is heavily promoted. Not unlike in the JAVA community, however, this is usually done in an ad hoc fashion, without an underlying theory when and how interfaces are to be used. While this question may have remained unobvious as long as programming languages such as C++ (which distinguish only between abstract and concrete classes, not between abstract classes and interfaces) were being used, with languages such as JAVA and C# the programmer must make an explicit choice, and this choice should be based on considerations reproducible by the readers of a program.

In the following, we elaborate on certain frequent patterns of interface-based programming, trying to sharpen the reader's awareness of the different uses of the interface concept, and provide programmers with robust criteria as to when one kind of interface or the other is being used. We start in Section 2 with a short survey of the evolution of interfaces as types (the indispensable prerequisite of interface-based programming), and define a few basic properties of interfaces in Section 3. Based on these definitions, Section 4 then provides a classification of interfaces together with the patterns of their use. A brief discussion of related work and a justification of our approach conclude our work.

2 EVOLUTION OF THE INTERFACE-AS-TYPE CONCEPT

Defined as "a shared boundary across which information is passed" [IEEE91], interface is a very general notion that has many uses in computer science. Interfaces between software components were pioneered by Dijkstra and Parnas in their work on software architecture and good design – the fact that it did not take long until interfaces became distinct programming entities in languages as respected as MODULA and ADA must be considered strong evidence for the import of the concept. Today it appears that the terms module and interface have almost become inseparable in software engineering: the ACM computing classification system for instance lists them jointly under *Design Tools and Techniques* (entry D.2.2).

With the advent of object-oriented programming, classes quickly became the primary unit of modularization, replacing to a large part for the more heterogeneous concept of a module. The interface of a class is an abstract data type (including formal behaviour specification) for which the class provides one (out of many possible) implementations. However, in most contemporary object-oriented programming languages interface specification is inseparably tied to the definition of a class (by tagging some of the class's features as public and others as private) – it can neither be shared by other classes (except of course by its subclasses), nor can the class implement more than one abstract data type (except those inherited from its superclasses). We call an

abstract data type that is part of the class definition the *implicit interface* of the class; implicit interfaces will play a role in some of our further considerations.

CLU was perhaps the first language to separate a class from its interface by way of associating a distinct type with each. While the primary reason for this was to permit separate compilation of modules (called clusters in CLU), it would also have made possible the selection among different implementations of an interface at run-time, as its authors note [Liskov77]. In fact, the full power of separating interfaces and implementation types is unfolded only when combined with type subsumption and the principle of substitutability: if instances of different types realize the same interface, they are allowed to replace for each other. Although more fundamental than the substitutability tied to the inheritance hierarchies of object-oriented programming languages, it took some time for so-called inclusion polymorphism to become decoupled from subclassing: the creators of JAVA confess to have borrowed the language's interface concept from OBJECTIVE-C (called protocols there), but the true origin is difficult to trace.[1]

C# has taken over the interface-as-type concept from JAVA and has developed it further. Most notably, C# lets classes offer different implementations of the same method if this method is declared in more than one of the class's implemented interfaces, and it allows interfaces to make accessible features of their implementing classes otherwise inaccessible (so-called explicit interface implementations). The latter is of particular interest since it allows the implicit interface of a class to be disjoint from the explicit interfaces it implements[2]; in fact, the implicit interface of a class can even be empty, forcing all variables providing access to the features of a class to be interface-typed. Both properties of C# are in line with Microsoft's strong emphasis of the interface-based programming paradigm [Pattison00], favouring interface over implementation inheritance.

By today, the interface-as-type concept has emancipated itself to the extent that whole software architectures (such as the OMG's CORBA or Microsoft's COM) are completely defined in terms of interfaces, not classes. The expected advantage of this is a better (dynamic especially) composability of software and – as has been right from the inception of the interface concept – an insensitivity to changes. However, in a world of interfaces only, one important aspect of interface-based programming is missed: the fact the same class can implement several, otherwise unrelated interfaces at the same time. Trivially, in total absence of classes the distinction between classes and interfaces as separate types disappears: it is reduced to the classical separation of specification and implementation.

Throughout the following, we think of an interface as a *named type specifying a (not necessarily complete) set of features characterizing all instances whose classes declare to*

[1]The language EMERALD [Raj91] is another possible role model for the introduction of interfaces as types in JAVA: in EMERALD, all variables must be interface-typed, and classes are completely replaced for by (implicitly typed) constructors.

[2]To be more precise, the implicit interface of a C# class does not automatically extend the interfaces the class implements.

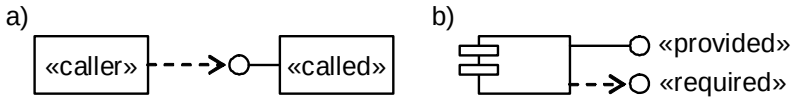


Figure 2. a) The two sides of an interface (notation is UML). b) Different interfaces.

implement the interface. Technically, "features" is commonly restricted to methods, but conceptually, they include attributes – access to which can be granted through accessor methods as part of the interface – as well.

3 BASIC PROPERTIES OF INTERFACES

In order to be able to describe the different patterns of interface utilization in Section 4, we need a certain vocabulary that covers the different properties of interfaces and their relationships to implementing classes. The list given below may appear random at first glance (and is certainly incomplete as regards the general properties of interfaces); yet it covers all we need to discern the different uses of interfaces as types.

Caller and Called: The Two Sides of an Interface

Every interface has two sides: it separates the *caller* and the *called*, the two roles of the asymmetric *calls* relationship. In UML jargon, the caller *depends* on the interface, and the called *implements* it (Figure 2 a). However, this naming is problematic, since the true dependency may sometimes be inverted (see below). Note that although caller and called are usually associated with classes, it is really objects that play the corresponding roles.

In the context of components, interfaces are usually classified as either *provided* or *required* (or *ingoing* and *outgoing*). Unlike caller and called, however, provided and required do not refer to two sides of the same interface, but to two different interfaces, by naming their roles with respect to a single component (Figure 2 b). Although both the provided and the required interface have a caller and a called side, these roles do not refer to the same components: in fact, the caller of a provided interface can be another component than the called of a required interface. Note that if dependency of two components is mutual, this has to be expressed by two opposing interfaces, with roles swapping depending on the interface being looked at.

In software engineering jargon the caller of an interface is often called the *client* and the called the *server*. This naming is biased towards a certain use of interfaces, though; in fact, calling in the above (technical) sense is not necessarily tantamount to requesting a service, at least not in the original sense of word (which would imply the benefit of the call to be on the side of the caller). Rather, as will be seen it may be the case that the caller calls the called for the latter's own profit, inverting the roles of client and server.

In an object-oriented program, callers of an interface are marked by variables declaring that interface as their type. The called on the other hand are the classes (or, rather, their objects) implementing the interface, be it directly or indirectly. Elsewhere we

have defined metrics counting the number of call sites (so-called *Interface Popularity*, IPOP) and implementations (so-called *Interface Generality*, IGEN) [Mayer03, Steimann03, Gössner04]; these metrics have been used to compute the numbers of Figure 1 and, as will be seen, they also provide important hints for the distinction between the different usage patterns of interfaces.

Interfaces and Contracts

An interface is sometimes regarded as the specification of a contract between a *client* and a *supplier* [Meyer97]. Both sides of the contract have *benefits* and *obligations*: the client is obliged to adhere to the preconditions associated with a needed service and benefits from the supplier ensuring the postconditions, whereas the supplier benefits from the client's keeping to the preconditions and is obliged to guarantee the postconditions. Although the names client and supplier suffer from the same problem as client and server mentioned above, the notions of benefit and obligation are somewhat more neutral; they will play an important role in our distinction of the different kinds of interfaces.

Preconditions and postconditions specify the functional (or behavioural) part of an interface. Depending on the particular kind of interface, they can be tight (as reflected in lengthy prose or complex logical expressions) or loose (as loose as constraining only formal parameter and return types of a single method). Generally, the laxer a contract is, the more freely a supplying class can behave. This goes as far as allowing the implementing class to do anything, as is for instance the case with the `Runnable` interface of the JDK. To the other extreme, interfaces specifications can be so tight that the only allowable variability in different implementations is in the non-functional requirements (such as time or space consumption). This variability is also an important criterion for the distinction of the different patterns of interface use as presented below.

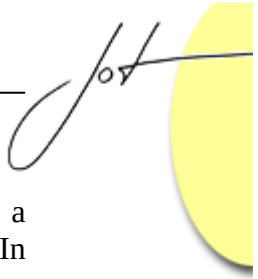
Total and Partial Interfaces

The classical interface separating specification from implementation comprises all public features of its implementing class(es). We call such interfaces *total interfaces*.[3] However, totality of interfaces is not always a desired property. Quite to the contrary – the Reference Model of the Open Distributed Processing standard (ODP RM) defines interface as follows:

> *Interface: An abstraction of the behaviour of an object that consists of a subset of the interactions of that object together with a set of constraints on when they may occur. Each interaction of an object belongs to a unique interface. Thus the interfaces of an object form a partition of the interactions of that object.* [ISO]

Although we find the strict partitioning (i.e., the non-overlapping of interfaces) required in this definition debatable, we agree that partiality of an interface (which can have different causes and may serve different purposes) is an important concept; we call such

[3] As noted above, an interface specifies features of objects, not of its implementing classes. Since constructors are class methods, the fact that an interface does not include constructors does not mean that it is not total.



interfaces *partial interfaces*. Note that whether or not an interface is partial is not a property of the interface alone, but equally determined by its implementing class(es). In fact, an interface can be both total and partial (but only of different classes). Thus, when speaking of a total interface without referring to a particular implementing class we require that it is a total interface of all implementing classes. It follows that absolute totality is not a property that can be attributed to interfaces of open systems, unless there are language means to prevent implementing classes from adding features.

Interfaces and Abstraction: Generalizations vs. Roles

By definition, an interface is always an abstraction: it abstracts from implementation by providing its specification.

In object-oriented programming, a supertype abstracts from its subtypes by omitting some of their features (so-called generalization). The interface of a generalized supertype is partial in the sense that it does not cover all public features of all its implementing classes. Induced by generalization, this partiality is the result of grouping together classes that share considerable commonality, differing only in some detail (with the opposite of generalization, specialization, adding the detail that distinguishes one subtype from the other).

Partiality does not necessarily result from generalization, however; instead, a partial interface may be designed to isolate a certain *aspect* or *facet* of its implementing classes. Although such a partial interface is still an abstraction, it does not omit detail, but instead focuses on it (by omitting all features that are not associated with the aspect). Partial interfaces of this kind reduce the generality of objects, by focussing on a specific use or appearance in a specific context. Such interfaces are *context-specific*; as elaborated elsewhere, they can be equated with *roles* [Steimann00b, 01b].

It is not always easy to distinguish a role from a generalization. In fact, even though there is a fundamental conceptual difference between the two, it is hard to tie it to formally observable properties, since the programmer – in expressing her/his intent – is free to (ab)use the programming language constructs at hand any which way (s)he pleases. One possibility is to assume that intent is expressed literally, i.e., that roles have role names (like *Customer*, *Printable*, etc.) and generalizations have *genera* names (like *Thing*, *Document*, etc.). However, there is no law to enforce this semantic naming convention, so that it cannot be relied on. Another possibility is to apply ontological criteria: if the interface is defined in the context of some relationship and access to an object through the interface is transient in nature, then it must be a role [Steimann 00a]. However, this requires a deep understanding of the use of an interface in a program, and that this use is stable across all possible extensions, which is seldom the case. Last but not least, one could argue that if an interface is partial, is not the only abstraction of the class, and is not extended by some other interface of the same class (excluding the implicit interface in the case of JAVA, since this interface automatically extends all implemented interfaces of the class), then it is a role. This seems justified because one must suspect that there are contexts in which the class is used differently than allowed by the interface (namely through the other existing abstractions), making the interface (which is not itself

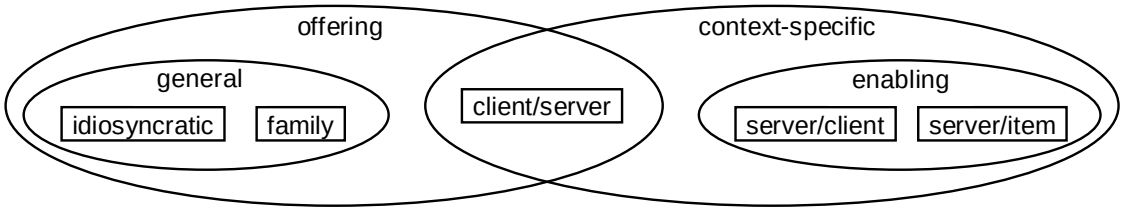


Figure 3. Classification of interfaces. *Offering* and *enabling* are complementary categories, as are *general* and *context-specific*.

a generalization of some other interface) context-specific. However, cases can easily be constructed where this (technical) criterion is misleading so that in practice, a combination of all three has to be applied.

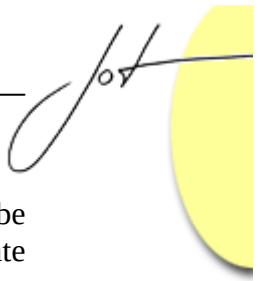
Note that as generalizations, interfaces compete with abstract classes. In fact, in practice interfaces and abstract classes are sometimes used as if they were the same concept. However, since abstract classes have the potential to pass on implementation to their subclasses, they should be used if (and only if) the relationship to the subclasses is genetic, i.e., if it is (or at least could be) based on the inheritance of internal structure, that is, implementation. If on the other hand the relationship is based on pure function (or, weaker still, on sameness of protocol), interfaces should be used. For instance, a linked list and a dynamic array would normally not be genetically related (i.e., have no common pieces of implementation), yet they share the interface of lists (specifying sequential access to their elements).

4 CLASSIFICATION OF INTERFACES

While the technical definition of the interface concept is unambiguous, its actual utilization can differ greatly. In trying to order the different usage patterns of interfaces, we have set up the classification shown in Figure 3. Although it has no strict hierarchical structure (it is not a tree, but built on two orthogonal subclassifications), the remainder of this section assumes a linear ordering, progressing through it from left to right and introducing the different uses along the way. The classification is based as much as possible on the terms and definitions presented in Section 3; one should keep in mind, though, that programming has many degrees of freedom, and any attempt to classify working code into a set of academic categories must either fail or suffer from a certain elusiveness.

Offering Interfaces

According to its standard conception, an interface publishes some service offered by the called to the caller. We call such interfaces *offering*. The benefit of the call to an offering interface is clearly on the side of the caller, whereas the greater part of the obligations are on part of the called. This is usually reflected in rather specific postconditions for the offered services, whereas the preconditions tend to be brief (and are formulated mostly in the server's terms). For instance, while a stack must not be empty in order to pop it (the



precondition), actually popping it requires that the most recently pushed element be removed, that the second be popped next, etc. (the postcondition). As an immediate consequence, there is usually a rather large number of potential callers to an offering interface (since the preconditions are easily fulfilled), whereas there is typically only a rather small number of implementors (because they must all fulfil the same specification, there will usually only be few alternative implementations).

Depending on whether an interface is intended for the general public or for specific clients, we distinguish between general and context-specific interfaces

General Interfaces

General interfaces are defined with no particular caller in mind. Because their purpose is to offer all services of their implementing classes to the general public, general interfaces are typically total. This is so because if not, one must suspect that there are contexts in which a class is used differently than allowed by the interface (as evidenced by the calling of methods excluded from it), making the interface context-specific.

Unless general interfaces are related by subtyping (i.e., one interface is a subinterface of the other), a class usually has only one general interface. This is so because there is no point in keeping apart interfaces that serve no specific caller or use. In fact, the only purpose of a general interface is to separate specification from implementation, making the latter (ex)changeable without affecting its callers. Depending on whether changes to the implementation reflect historical or concurrent (competing) alternatives, we further divide general interfaces into idiosyncratic and family interfaces.

If a general interface is implemented by only one class (whose implementation may be modified over time, but with no two alternatives occurring in the same project), we call this interface *idiosyncratic*.[4] Idiosyncratic interfaces are often named after the classes they specify the interface of, with either the interface or the class name being complemented by a prefix or suffix indicating the interface or implementor status, as in `IPerson` or `StackImpl`.

> ***Idiosyncratic Interfaces.*** *An idiosyncratic interface use is characterized by a to-one relationship of a total interface with its implementing class: only one class implements the interface at a time. (The class may however implement other interfaces, but these interfaces will typically be context-specific.) Because the interface is a complete abstraction of the class it implements, it can replace for the class in variable declarations.*[5] *This is the classical use of the interface-as-type concept.*

[4]Note that *general* relates to the caller's side of the interface (indicating that it is not aimed at a specific caller), whereas *idiosyncratic* pertains to the implementor's: only one class has this interface. Hence, there is no contradiction in calling a general interface idiosyncratic.

[5]Note that if language peculiarities prevent an idiosyncratic interface from being a complete abstraction of the class it represents, it cannot ubiquitously replace for that class in variable declarations. In the case of JAVA, if direct access to the fields or non-public or static features of a class is required, this prohibits the use of an interface in the class's place.

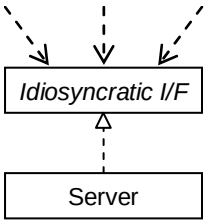


Figure 4. Idiosyncratic interface

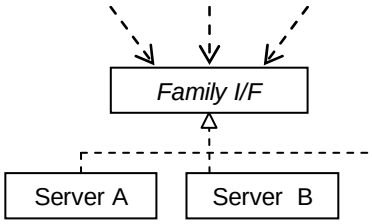


Figure 5. Family interface

Figure 4 illustrates the use of an idiosyncratic interface: a single class, the server, implements a total interface on which its (unspecified) clients depend. Because the clients are unspecified, later introduction of new clients (with perhaps new uses of the server) do not require new interfaces; the idiosyncratic interface is completely unspecific, that is, general.

In contemporary languages such as JAVA and C#, idiosyncratic interfaces will mostly remain implicit, because the set of features declared by a class as public can be thought of as its interface (the implicit interface; *cf.* Section 3). Factoring out theses features into a separate syntactic entity may not seem worth the effort, particularly since it is implicitly done by the compiler. Therefore, we cannot expect to find many idiosyncratic interface declarations in JAVA or C# program corpora. This is in contrast to the instructional literature on object-oriented programming, in which idiosyncratic interfaces (such as `IDog` or `IPerson`) abound.

As it turns out, there are no true examples of idiosyncratic interfaces in the JDK – the implementations of literally all interfaces implemented by only one class are tagged as examples. Things are different for ECLIPSE, however, which has a use for idiosyncratic interfaces even in JAVA programming: its `IClassFile` for instance is the interface of only one class, `ClassFile`, of which it is a total interface. To quote the documentation, this interface "represents an entire binary type (single .class file)" and is "not intended to be implemented by clients". In fact, `ClassFile` resides in a package explicitly marked as being internal to the system (`org.eclipse.jdt.internal.core`), contrary to its interface, `IClassFile`, which is from the publicly accessible package `org.eclipse.jdt.core`.

If a general interface is implemented by more than one class at the same time (i.e., within the same project), we call this interface a *family interface* (Figure 4). The different classes are often offered as alternative implementations, with different (technical) properties; yet each one adheres to the same interface specification. As opposed to idiosyncratic interfaces whose implementation can be varied only at design time of the program, family interfaces offer implementation alternatives at run time; they are therefore often accompanied by factories [Gamma95] and double dispatch as a substitute for multiple dispatching [Steimann01a].

> ***Family Interfaces***. *A family interface heads a family of classes, offering their services to the general public (i.e., unspecified clients). If the classes comprised under the family interface extend each other, it will be a generalization of some of the classes, but these usually have their own (idiosyncratic or family) interfaces.*

Because family interfaces are context-independent, they specify the nature of their objects rather than their behaviour in a specific role (*cf.* Section 3). Family interfaces therefore often carry typical class names (such as `Number` or `Interval`) and are used interchangeably with abstract classes as *patres familiae* of a family of classes [Steimann01a]. In fact, in absence of multiple (class) inheritance interfaces are sometimes abused (as regards the criteria of Section 3) to root class hierarchies where abstract classes would be in place, in order not to block the possibility of inheritance from other classes.[6]

Prominent family interfaces are `Collection`, `Set`, `List`, and others from the JAVA 2 collections framework. Note that `Enumeration` and `Iterator`, which are mostly implemented by anonymous classes[7] and by classes inner to the corresponding collections, are also family interfaces: although conceptually both interfaces are context-specific (providing sequential access to collections in the context of iteration), technically they are not – there is no other use of enumeration and iterator objects.

Because family interfaces are often only partial interfaces of some of their implementing classes, the demarcation from client/server interfaces (as discussed next) is sometimes difficult. For example, an instance of class `TreeSet` may be used as a plain collection in some places and as a sorted set in others, making its use context-specific. However, since `Collection` is a generalization (of both `SortedSet` and `TreeSet`), it does not focus on specific properties; hence, it will usually not be seen a role.

Context-Specific Interfaces (Roles)

An interface that is not general is context-specific; it comprises the specific protocol expected by certain callers whose relationship to the called (the implementors of the interface) sets up the context of their (the implementors') use. Because specificity is a property defined only in presence of variety, context-specific interfaces are typically partial. However, they may overlap.

Although it is conceivable that context-specific interfaces are also idiosyncratic (i.e., the interface of only one class), one general theme behind context specificity is to minimize coupling, allowing a greater number of classes to take the place of the called. A distinction based on the count of implementors of a context-specific interface (as done for general interfaces) is therefore not useful. Instead, there is an interesting twist to the dependency relationship, differentiating offering from enabling interfaces (Figure 3).

For the obvious kind of context-specific interfaces, the caller relies on some specific service offered by the implementor of an interface: being the beneficiary of the interaction, it is the client of the service. On the other side of the interface, the called

[6] In fact, the JAVA API specification is full of Freudian slips confusing classes and interfaces: for example, the interface `MenuContainer` is defined as "the *super class* of all menu related containers", and `Streamable` as "the *base class* for the Holder classess of all complex IDL types".

[7] JAVA's anonymous class mechanism allows the ad-hoc creation of instances whose protocol conforms to an interface. If this interface is exclusively "implemented" by anonymous classes, then it designates the only context in which the instances can be used, and it is necessarily total so that it must be classified as a family interface. (It is not idiosyncratic because arbitrarily many different alternative implementations can be provided, as is the case for `Enumeration`.)

behaves as the benefactor: it is the server of the service (Figure 6). We therefore call such interfaces *client/server interfaces*. As for general interfaces, client/server interfaces usually come with tight contracts so that the called can often be substituted without affecting the caller's or global program behaviour (*cf*. Section 3).

> ***Client/Server Interfaces.*** *A client/server interface is a context-specific interface that offers the particular services needed by certain clients. The server typically has other features not included in the client/server interface; it offers different interfaces to different clients.*

Although client/server interfaces will typically be found in closed application programs in which both client and server have been designed specifically to interact with each other, they can specify partitioned access to general purpose classes as well. For instance, an interface `Stack` could be used to specify access to an instance of class `Vector` in all contexts in which it is used as a stack, without prohibiting the use of the same or other instances of the same class as something else (a queue for instance) in other places.

A typical example of a client/server interface is the use of the `MenuContainer` interface from the JDK (package `java.awt`). The interface provides functions for "all menu related containers"; it is implemented by classes as different as `Button`, `Checkbox`, `Scrollbar`, and `TextFrame`, all of which can – despite their different nature – present as menu containers to their clients. Clients rely on this particular property by accessing their servers through the `MenuContainer` interface; the classes implementing `MenuContainer`, however, have many other services to offer.

Other prominent client/server interfaces identified in the JDK are `Shape` and `LayoutManager` (both from `java.awt`), `DataInput` and `DataOutput` (from `java.io`), and `java.lang.CharSequence`. Interestingly, some of these interfaces would appear to be typical family interfaces: whereas *DataInput*, *DataOutput*, and *CharSequence* qualify as role names (names given in a certain context), *Shape* and *LayoutManager* sound like the names of natural types or generalizations (and hence would be expected to be family interfaces; *cf*. the argumentation of Section 3). However, the classes implementing `Shape` are really quite heterogeneous, with `Shape` specifying only one aspect common to all of these; not as consistently, but in a similar vein, `LayoutManager` is an interface of user interfaces, editors, and layout managers, focusing only on the layout manager aspect of each.

Enabling Interfaces

For general and client/server interfaces (the interface uses presented so far), the class implementing the interface is the service provider. However, as indicated above this need not always be so. In fact, there is an important category of interfaces in which the caller of the interface is the service provider (server) and in which the called is either the client

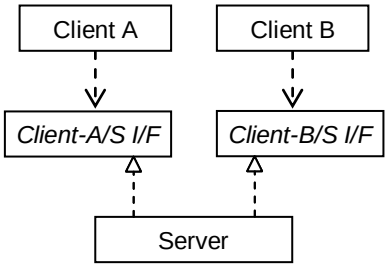


Figure 6. Client/server interface. Server offers its services to different clients, each with its own client/server interface, comprising only what is needed.

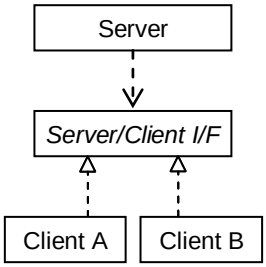


Figure 7. Server/client interface. The receivers of the service are the implementors of the interface.

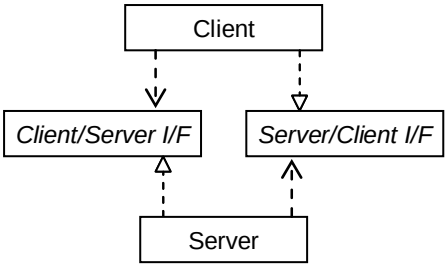


Figure 8. Server/client interface. The receivers of the service are the implementors of the interface.

or an item of it. Because these interfaces enable the called object's individual contribution to or participation in the caller's context, we call them *enabling interfaces*.

Enabling interfaces often carry an *-able* or *-ible* suffix, as for example `Printable`, `Accountable`, or `Accessible`. Characteristically, postconditions of enabling interfaces are not as tight as those for offering interfaces, so that exchanging one implementor for another will most likely alter program behaviour. In fact, different implementing classes of an enabling interface are typically carriers of different, application-specific code. This is in contrast to the caller, which will often be general purpose and can be substituted accordingly.

Depending on whether the called object or some third party is the beneficiary (client) of the service, we have subdivided enabling interfaces further, namely into *server/client* and *server/item interfaces*.

With server/client interfaces, the beneficiary is the called object, which profits from a service offered by the caller of the interface (the roles of client and server thus being swapped, as in Figure 7). As an immediate consequence, the contract for the implementing class is usually rather lax (as reflected by a fairly small number of methods required); that of the caller remains mostly implicit.

> ***Server/Client Interfaces.*** *A server/client interface is an interface that enables its implementors to profit from some service offered by the caller. A server/client interface is often specific to a particular server; however, it need not be.*

Not coincidentally, the naming suggests that server/client and client/server interfaces oppose each other: if interaction between client and server is bidirectional (with roles of the participating object being independent of the particular direction), these interfaces are likely to occur in pairs (Figure 8). Such is typically the case in asynchronous communication, when callbacks are required to return the value of a computation; however, it also occurs when the server needs additional information from the client not provided with the initial service request.

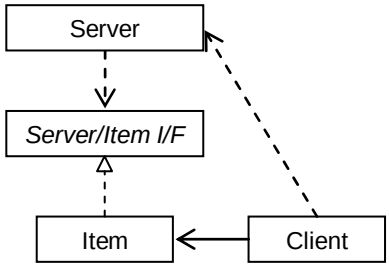


Figure 9. Server/item interface

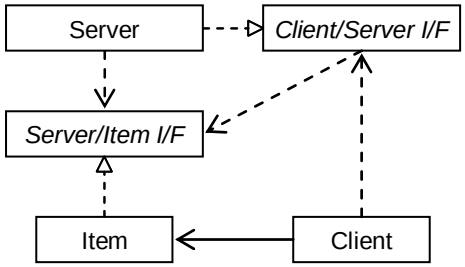


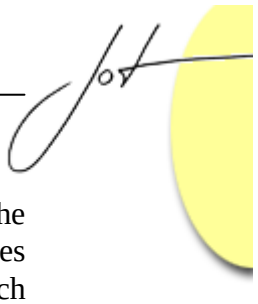
Figure 10. Service relying on a server/item interface, made available through a client/server interface.

Although paired client/server-server/client interfaces are common, server/client interfaces exist in their own right. For instance, the observer role of the OBSERVER pattern [Gamma95] is represented by a server/client interface implemented by the beneficiary of the collaboration, namely the object to be notified when a change occurs. Typical server/client interfaces from the JDK are thus its listener interfaces. Other typical instances of server/client interfaces occur in frameworks, where the execution of user-provided classes (so called *plug-ins*) is controlled by a set of framework classes (a condition referred to as *inversion of control* [Fayad97]). Being enabling interfaces, it is the implementing class (the client) which provides the application-specific code; the calling class (the server) on the other hand is typically application independent (for application frameworks: independent of a specific instantiation). Note that although server/client interfaces can in principle be total, they are usually only partial since the plug-in classes need to interact with others to fulfil their application-specific purpose, of which the server has no knowledge.

The prototypical example of a framework server/client interface is the `Runnable` interface of the JDK (package `java.lang`), providing for multithreading in JAVA. A class implementing `Runnable` does this because it wants to run a separate (its own) thread so it can act independently of others, and it receives this thread (the service) from `Thread`, the class calling the `Runnable` interface. The contract for `Runnable` is minimal: it consists of a single method `run` with no arguments. Even semantically, this method is totally unconstrained: to quote the JDK documentation, "[t]he general contract of the method run is that it may take any action whatsoever." Other prominent server/client interfaces with similar protocol are `Action` and `MenuElement`, both from the SWING GUI framework.

The relationship between the server and the client of a server/client interface is typically rather long-lasting. As it turns out, server/client interfaces can often be identified by the fact that the server offers a special procedure for registering its clients (and a corresponding attribute keeping references to them). This holds particularly for plug-ins and observers; it does not, however, apply to the links held by a server calling back its clients, which are typically temporary in nature.[8]

[8]Note that in the common type system (CTS) of Microsoft's .NET framework, both callbacks and observers are realized by so-called delegates, special types representing function pointers.



For many enabling interfaces the beneficiary of the service is not the implementor (the called) itself, but some other class holding it. Typical examples are the *-able* interfaces `Comparable` and `Accountable`; for instance, it is not the comparable object which profits from being compared, but the object holding it, as for example a collection that is to be sorted (or, ultimately, the owner of the collection). Because the implementing object is typically an item of some other class, we call these interfaces *server/item* interfaces.

Server/item interfaces occur in collaborations of three or more objects in which one object (the item) is passed by another (the client) to a third (the server) which is to process the item for the client (Figure 9). This processing requires some support from the item, which offers this support by implementing the server/item interface. As opposed to server/client interfaces, the relationship established between the caller (the server) and the called (the items) of a server/item interface is temporary in nature. Therefore, server/item interfaces typically occur as the types of formal parameters and temporaries, but not of attributes (fields).

> ***Server/Item Interfaces***. *A server/item interface is an interface that enables the processing of a certain kind of objects (the items) by a server for the profit of some third party, the client.*

Server/item interfaces are frequently found in conjunction with offering (client/server especially) interfaces. The offering interface then includes the server/item interface as a formal parameter type (Figure 10). Note that in the example of sorting collections, this would require a generic type (a `List` of `Comparable`s).

Other prominent server/item interfaces are `Printable` (from `java.awt.print`), `LazyValue` (inner to `javax.swing.UIDefaults`) and `XMLWritable` (from `org.apache.crimson.tree`). Note that in the case of `Printable`, one might argue that it is the printable object itself that profits from being printed. However, unless the object and the printer are related somehow (which is typically not the case for implementors of `Printable`), some third party must be the issuer (and beneficiary) of the printing request.

Special cases of server/item interfaces are the so-called tagging or marker interfaces.[9] Tagging interfaces are often empty, in which case they can only occur as parameter (and not as receiver) types of method calls. The dependency arrow from the server to the server/item interface in Figure 10 then stands for a dynamic type check (usually in the form of an instance-of test), the primary purpose of a tagging interface.[10] On the other hand, many tagging interfaces are abstract in the sense that they are not directly implemented, but root a hierarchy of non-empty interfaces, or they extend non-empty interfaces without adding functionality.

[9]The names are to express that the so-labelled interfaces flag their implementing classes as being of a certain type. This information can be utilized by the compiler (through type checking) as well as during program execution (by querying an object's conformance to the interface). Quite obviously, tagging interfaces share this property with all others.

[10]Tests for identity are another possible operation for tagged objects. Characteristically enough, however, the JDK's most frequently implemented empty interface, `Serializable` (implemented 1975 times in the JDK 1.4) grants access to all attributes of its implementing classes, albeit only through introspection.

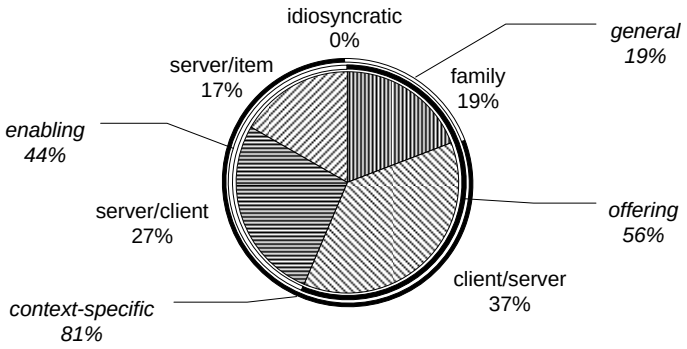


Figure 11. Relative distribution over the five different uses.

Relative Distribution of Interfaces

To probe the relative frequency of the different kinds of interfaces, we looked at the 100 most often implemented and at the 100 most often referenced (as counted by the number of variables) interfaces from the JDK 1.4.[11] Between the two groups, there was an overlap of 43. One interface contained only constants; we excluded it from our list because it did not serve as a type.

Among the remaining 156 interfaces, there are no idiosyncratic and only 30 family interfaces (together comprised as general interfaces; of these, only four are total interfaces of all their implementing classes); the remainder (126) is context-specific, i.e., classified as roles. Among these, less than half (56) are client/server interfaces; the rest (68) was categorized as enabling interfaces, namely 42 as server/client and 26 as server/item. Among the 42 server/client interfaces 22 are listeners.

As can be seen from Figure 11, offering (i.e., family and client/server) interfaces dominate over enabling interfaces, but this dominance is not as clear as one might have expected. This may be due to our selection criteria, which placed equal weight on number of implementations and number of variables (or call sites; *cf.* Footnote 11). In fact, more than half of the most often implemented interfaces were classified as enabling, compared to only one third of the most often referenced (which lead in the offering category). This should come as no surprise, though, since clients are likely to be more common than servers, meaning that offering interfaces should be referenced more often than they are implemented, whereas enabling interfaces (with roles of client and server inverted) should be implemented more often than they are referenced.

Investigating this relationship further, Table 1 reveals that average values for IGEN and IPOP as well as their quotient distinguish pretty well between offering and enabling interfaces: in fact, on average there is less then one variable per implementing class of an enabling interface, while there are almost four per class for each offering interface. Also,

[11]We classified all interfaces manually, by looking at the source code and documentation. Quite clearly, there was no way of doing this for all of JDK 1.4.1's 857 interfaces. In order to include the most interesting interfaces and at the same time keep the bias small, we decided to pick those with the highest numbers of implementations and variables, respectively. However, as it turned out the ratio of these counts separates offering and enabling interfaces pretty well, so that there is a certain bias towards an equal distribution of our sample between these two groups.

Table 1: Average number of implementations (IGEN), variables (IPOP), and methods for each category of interfaces.

CATEGORY	∅IGEN	∅IPOP	∅(IPOP/IGEN)*	∅NUMBER OF METHODS
: : idiosyncratic	–	–	–	–
: : family	22	79	3.17	20
: : client/server	21	63	4.31	19
: offering	21	69	3.86	19
: : server/client	54	34	0.91	4
: : server/item	134**	40	0.41	7
: enabling	84	37	0.73	5
Overall	49	55	1.91	12

* computed as the geometric mean

** high value explained by `Serializable`'s IGEN being 1975; without it, ∅IGEN would be 60 and thus comparable to that of server/client interfaces.

the average size of the interfaces (as measured by the number of methods) is significantly larger for offering than for enabling interfaces; this is in accord with the assumption that the contract for enabling interfaces is much weaker than for offering (general and client/server) interfaces (*cf.* Section 3). Note that average interface size does not differ much between family and client/server; while this may be partly due to the fact that the boundary between these two in open general purpose libraries such as the JDK is rather blurry, we have observed elsewhere [Mayer03b] that client/server interfaces tend to be not as context-specific (narrow) as they could be, including more methods than actually needed.

We would have liked to present a table analogous to Table 1 here with measures for formally differentiating general from context-specific interfaces, but were unable to identify robust decision rules. This is in line with the already mentioned (in Section 3) lack of sharp criteria distinguishing generalizations from roles.

5 DISCUSSION

Related Work

The general literature on object-oriented software engineering and programming strongly advocates the use of interfaces. However, little is written about how interfaces are to be identified and introduced systematically. Coad's book about the design of JAVA applications [Coad99] and one about programming with COM [Pattison00] are noteworthy exceptions; note that the latter is also one of the few sources using the term *interface-based programming*.

In his lengthy treatment of the design with interfaces, Coad has identified mainly two different categories: "the *kinds of classes* whose objects you want to plug into that plug-in point", and "the *kinds of behaviour* you want such objects to exhibit". In the latter category, interfaces contain "little groupings of functionality" within a broader "kind of class" classification. This category is subdivided into interfaces indicating an algorithm's

plug-in points (enabling interface in our terminology) and interfaces indicating that a so-called feature sequencer is expected at a given point. [Coad99]

Note that although Coad discusses the relation of roles to interfaces, he views roles as suggested by the ROLE OBJECT PATTERN [Bäumer97]. By contrast, we have equated roles with context-specific interfaces [Steimann00b, 01b], a standpoint that is supported by the fact that these interfaces often carry role names (e.g., the *-ables* and *-ibles*), and that patterns themselves are defined in terms of roles (including the ROLE OBJECT PATTERN, which cannot be applied to itself) [Steimann00a].

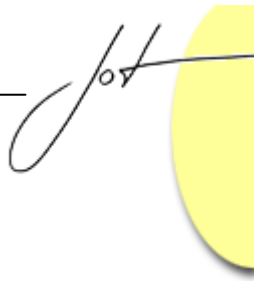
The literature describes other programming languages that emphasize interfaces over classes. For instance, EMERALD has no classes (only constructors); it uses interfaces as types in variable declarations [Raj91]. Type checking in EMERALD (as required for variable assignments) is performed statically wherever possible; where not, a run-time type check is inserted and executed dynamically to ensure that further operations (method calls) are safe. However, since interfaces are the only kind of type in EMERALD, there are no interface-specific usage patterns to be observed.

Why JAVA and the JDK?

We chose to look at JAVA programs because the JAVA language (like C#) offers both abstract classes and interfaces as syntactically distinct constructs, giving the programmer the opportunity of being explicit about her/his intentions: although completely abstract classes and interfaces can be (and sometimes are) used interchangeably to a certain extent, we expect the resultant classification errors to be small, especially when compared to the errors induced by manually deciding whether an abstract class was conceptually intended to be an interface or the root of a class hierarchy, as we would have had to do, for instance, with C++ class libraries.

We selected the JDK mainly because it is rather well-known to a wide audience, because it is well-documented, and because an archive of older versions is available. We have also looked at other large and freely available packages; of these, ECLIPSE appears to make the most disciplined use of interfaces, but its API is known only to a smaller audience and examples would have required much more explanation.

Note that it is generally difficult to classify the interfaces of an open API such as the JDK as we did, because utilization of the offered types in actual applications is somewhat unpredictable. However, since the JDK is also a framework making extensive use of its own classes and interfaces, we are confident that the derived numbers have some practical relevance.



6 CONCLUSION

While there seems to be a certain consensus that the introduction of interfaces as types syntactically distinct from classes is a good idea, only little work has been spent on the investigation of their practical use. The reason for this apparent lack of interest may be that the concept seems too simple to be subject to examination. On the other hand, we could identify a number of fundamental properties distinguishing different kinds of interfaces, and derive guidelines for their systematic use. A set of five basic patterns of interface utilization together with figures indicating the frequency of their occurrence are the results of our work.

REFERENCES

[Bäumer97] D Bäumer, D Riehle, W Siberski, M Wulf "The role object pattern" in: *PLoP ´97 – Proceedings of the 1997 Conference on Pattern Languages of Programs* (1997).

[Coad99] P Coad, M Mayfield *JAVA Design: Building Better Apps and Applets 2nd edition* (Yourdon Press, Upper Saddle River 1999).

[Fayad97] ME Fayad, DC Schmidt "Object-oriented application frameworks" *CACM* 40:10 (1997) 32–38.

[Gamma95] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns – Elements of Reusable Software* (Addison-Wesley, 1995).

[Gößner04] J Gößner, P Mayer, F Steimann "Interface Utilization in the Java Development Kit" in: *Proceedings of the 19th ACM Symposium on Applied Computing SAC 2004* (Nicosia, Cyprus, 2004) 1310–1315.

[IEEE] IEEE *Standard Computer Dictionary* (IEEE, 1991).

[ISO] ISO/IEC *Open Distributed Processing – Reference Model – Part 2: Foundations* International Standard 10746-2 /ITU-T Recommendation X.902

[Liskov77] B Liskov, A Snyder, R Atkinson, C Schaffert "Abstraction mechanisms in CLU" *CACM* 20:8 (1977) 564–576.

[Mayer03a] P Mayer *Eine Metrik-Suite zur Analyse des Einsatzes von Interfaces in Java* Bachelor Thesis (Universität Hannover, September 2003).

[Mayer03b] P Mayer "*A*nalyzing the Use of Interfaces in Large OO Projects". in: *OOPSLA 2003 Companion* (Anaheim, USA , October 26-30, 2003) 382–383.

[Meyer97] B Meyer *Object-Oriented Software Construction* Second Edition (Prentice Hall, 1997).

[Pattison00] T Pattison *Programming Distributed Applications with COM & Microsoft Visual Basic* (Microsoft Press, 2000).

[Raj91] RK Raj, ED Tempero, HM Levy, AP Black, NC Hutchinson, E Jul "Emerald: A General-Purpose Programming Language" *Software – Practice and Experience* 21:1 (1991) 91–118.

[Steimann00a] F Steimann "On the representation of roles in object-oriented and conceptual modelling" *Data & Knowledge Engineering* 35:1 (2000) 83–106.

[Steimann00b] F Steimann "A radical revision of UML's role concept" in: E Evans, S Kent, B Selic (eds) *UML 2000: Proceedings of the 3rd International Conference* (Springer, 2000) 194–209.

[Steimann01a] F Steimann "The family pattern" *Journal of Object-Oriented Programming* 13:10 (2001) 28–31.

[Steimann01b] F Steimann "Role = Interface: a merger of concepts" *Journal of Object-Oriented Programming* 14:4 (2001) 23–32.

[Steimann03] F Steimann, W Siberski, T Kühne "Towards the systematic use of interfaces in JAVA programming" in: *Proceedings of 2nd Int. Conf. on the Principles and Practice of Programming in JAVA* (Kilkenny, 2003) 13–17.

About the authors

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

Philip Mayer is a graduate student of Computer Science at the Universität Hannover. In his Bachelor's Thesis he developed an extensive metric suite measuring the use of interfaces in Java programs. He is one of two developers of intoJ, an open source Eclipse plugin analyzing the access of types. He can be reached at plmayer@acm.org.