Interface Utilization in the JAVA DEVELOPMENT KIT

Jens Gößner Philip Mayer Friedrich Steimann

Institut für Informationssysteme
Fachgebiet Wissensbasierte Systeme
Universität Hannover, Appelstraße 4
D-30167 Hannover

{goessner, mayer, steimann}@kbs.uni-hannover.de

ABSTRACT

Interfaces as defined in the JAVA programming language can enhance both decoupling and comprehensibility of large code bases. Several researchers have pointed out this key role of interfaces in object-oriented programming, but so far only little insight as to how interfaces are actually used in practice has been made available. We fill this gap by applying a special metrics suite to one of the most popular pieces of software, the JAVA DEVELOPMENT KIT, and present interesting results.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Abstract data types, Classes and objects, Data types and structures, Frameworks, Inheritance, Patterns, Polymorphism*

General Terms

Measurement, Design, Experimentation, Languages.

Keywords

Metrics, Interfaces, JAVA, Frameworks, Refactorings.

1. INTRODUCTION

"*Program to an interface, not an implementation*" [1].

The extensive use of interfaces in object-oriented programming can enhance both comprehensibility and modularity of the code. However, as recent research has shown [2][3], interfaces are not very popular among programmers, not even of large and widely visible code bases. As has been point out, the reason for this may lie in the additional expenditure of time needed for the introduction and maintenance of interfaces, and in the failure to understand how to use interfaces in the first place.

In [2], a set of refactorings towards a better use of interfaces has been proposed together with a suite of interface-related metrics measuring the actual utilization of interfaces in object-oriented programs, but neither a precise definition nor an implementation has been provided. In this paper, we formalize the metrics proposed in [2], comparing them to the better known fan-in/fan-out metrics where possible, and make corrections where necessary. We then apply these metrics to one of the world's most popular code bases, the JAVA DEVELOPMENT KIT (JDK)[1], and present the results in tabular form, discussing our findings and the insights gained. As it turns out, interfaces are used in the JDK for many different purposes, but not always as consistently and to the extent one might have expected. By investigating the effects of a simple sequence of refactorings on the relevant metrics we show how the use of interfaces can be increased even after a program has been written.

SAC'04, March 14-17, 2004, Nicosia, Cyprus

2. RATIONALE

For the following we have chosen a JAVA project because JAVA (like C#, but unlike C++ or SMALLTALK) lets the programmer be explicit about a given type being an abstract class or an interface. We have chosen the JDK because we assume that its contents are known (and our results will be meaningful) to a wide readership. Results are biased to a certain extent by the JDK's being a class library; however, for large parts it is also a framework (a semi-complete application) and as such representative of many code bases in use. Last but not least, some of the JDK's frameworks (as for example the collections framework) are heavily used from within the JDK, so that it is really a mixture of a class library, a framework, and an application. Generalizing the results presented here to current programming practices is another story; for the time being, we encourage all readers interested in their use of interfaces to apply the metrics we provide to their programs.

3. INTERFACE STATISTICS

3.1 The JDK's most general interfaces

An interface can be very specific in the sense that only few classes implement it. Conversely, we can say that the more often an interface is implemented, the more general are the features offered by this interface. To measure this fact, a metric called *Interface Generality* has been defined as follows:

"Generality of an interface measures its dissemination defined as the number of classes implementing it. The more classes implementing an interface, the more general this interface may be assumed to be. If there is only one class implementing the interface, this indicates that it is rather special." [2].

[1] The version analyzed was J2SE 1.4.1_02.

Formally, the definition of Interface Generality (*IGEN*) translates to

$$IGEN_I = \sum_{i=1}^{n} \begin{cases} 1 \Leftrightarrow C_i \leq I \\ 0 \quad else \end{cases}$$

where C_i ranges over all classes in a project and $C_i \leq I$ stands for class C_i implementing interface I (directly or indirectly, either through extending superclasses or through implementing subinterfaces). IGEN is equivalent to the inheritance fan out factor of an interface — where inheritance fan out generally measures the number of subtypes of a chosen type — only that subtypes are restricted to classes (thus excluding subinterfaces) [4].

Table 1 shows the most general interfaces of the JDK. Not surprisingly, the interface implemented most often is `Serializable` (22.8% of all interface implementations). `Serializable`, like `Clonable` and other empty interfaces, are sometimes called marker or tagging interfaces [5][6].

Next comes `java.util.EventListener`, which introduces a second family of interfaces extensively implemented: the Listener interfaces. Listeners are the Observers in the Observer pattern [1] which is heavily used in the AWT and Swing implementations. `EventListener` — which is also called a tagging interface in the JAVA documentation [5] — is subtyped by `ActionListener` (4th) and `ImageObserver` (6th).

Tagging interfaces are called tagging because they classify the implementing class as being of a certain type.[2] More characteristically, they enable the use of the class's instances in a specific context. Therefore, we prefer to call these interfaces *enabling interfaces* [7]. However, enabling interfaces need not be empty: for instance, by specifying `compareTo(.)` the `java.lang.Comparable` interface enables the ordering of the instances of its implementing classes. Likewise, `Runnable` lets instances of its implementing classes run their own thread of execution, simply by implementing `run()`.

Enabling interfaces are often recognized by their suffix "-able" or "-ible". However, the Listener interfaces are also enabling interfaces, since they enable the implementing classes' participation in the notification procedure implemented by the Subject of the Observer pattern [1]. Clearly, enabling interfaces dominate Table 1: supplying 111 out of 829 interfaces (13.3%) in the JDK, these two groups account for 53.7% of the total number of interface implementations (30.9% without `Serializable`).

Table 1. The most general interfaces of the JDK

#	Name of Interface	IGEN	IPOP[3]
1	java.io.Serializable	1975	140
2	java.util.EventListener	584	92
3	java.lang.Cloneable	535	0
4	java.awt.event.ActionListener	235	79
5	javax.accessibility.Accessible	210	194
6	java.awt.image.ImageObserver	209	43
7	java.awt.MenuContainer	209	8
8	org.omg.CORBA.portable.IDLEntity	183	1
9	javax.swing.Action	178	175
10	java.security.PrivilegedAction	158	8

[2] Of course, they share this property with all other interfaces.

[3] IPOP stands for *Interface Popularity* (subject of Section 3.2).

3.2 The JDK's most popular interfaces

While generality of an interface considers the supplier's view, i.e., how many classes offer the services specified by an interface, the client side also deserves some attention. The question here is: How many times is an interface type used/referenced by a client for a variable declaration? The more references to an interface type occur in a project, the more *popular* this type may be thought to be:

"Popularity of an interface counts the number of variables declared with that interface as their type. The higher the popularity, the more use is made of the interface; the greater is the number of contexts in which it appears" [2]. *Interface Popularity* (*IPOP*) can be formalized as follows:

$$IPOP_I = \sum_{i=1}^{n} \begin{cases} 1 \Leftrightarrow V_i : I \\ 0 \quad else \end{cases}$$

where V_i ranges over all variables (i.e. fields, temporary variables and formal parameters) in a project and $V_i : I$ means that the variable V_i is declared with interface I.[4] Like IGEN, the IPOP metric can be expressed as a fan factor: IPOP is equivalent to the coupling fan in factor of the interface, with coupling being defined as the number of variables declaring the interface as their type. [5]

Table 2. The most popular interfaces in the JDK

#	Name of Interface	IPOP	IGEN
1	org.w3c.dom.Node	715	65
2	javax.swing.text.Element	525	8
3	javax.swing.text.AttributeSet	486	20
4	java.util.Iterator	446	49
5	java.util.Enumeration	423	36
6	javax.swing.Icon	379	65
7	org.omg.CORBA.Object	364	77
8	java.awt.Shape	316	33
9	java.util.Map	248	30
10	java.util.List	234	29
…			
13	java.util.Set	197	29
…			
18	java.util.Collection	158	69
…			
29	java.lang.CharSequence	95	14

The most popular interface in the JDK is `org.w3c.dom.Node`, which is „the primary data type for the entire Document Object Model"[5]. Instances of this type represent a node in a XML-document. Interestingly, at rank two there is another interface that represents a part in a document model: `javax.swing.text.Element` describes "a structural piece of a document. It is intended to capture the spirit of an SGML element" [5].

The interfaces ranked 3rd and 6th–8th (`javax.swing.text.AttributeSet`, `javax.swing.Icon`, `org.omg.CORBA.Object`, `java.awt.Shape`) have something in

[4] Return types could be included as well, but have been omitted here since we have focused on the use of variables.

[5] In general, coupling fan factors measure the degree to which types depend on one another.

common, too: they each form the head or root of a family of classes [7]. They could equally have been represented by abstract classes, but were presumably designed as interfaces because of JAVA's lack of multiple (implementation) inheritance. In fact, in JAVA "[t]he `org.omg.CORBA.Object` interface is the root of the inheritance hierarchy for all CORBA object references in the Java programming language" [5]. By contrast, in C++ `CORBA:Object` it is a class from which all other CORBA-classes inherit implicitly: "The `Object` class is the base class for all normal CORBA objects." [8].

4th in our list comes `java.util.Iterator`, followed immediately by `java.util.Enumeration`. Both interfaces can really be considered as one, since they share the same purpose: iteration over collections. Taken together, the two are the most popular interfaces in the JDK (referenced 869 times). Of course, the reason for this is the absence of a built-in iterator construct in JAVA, an issue that is hoped to be remedied in future versions. Table 2 may be considered evidence that such a fix is truly needed.

The top 10 of the most popular interfaces is concluded by `java.util.Map` and `java.util.List`, which stem from the JAVA collections framework. This raises the question how often the other collection interfaces are referenced. In fact, accumulated all collection interfaces (Appendix A) would be ranked first, referenced 1785 times. Even more revealingly, although only 1.44% of all interfaces in the JDK are collection interfaces the combined popularity of them is 12.68%. Note that like `Icon`, `Object`, and `Shape`, the collection interfaces are roots of small families of classes, too.

It is instructive to note that the top 10 of the most popular interfaces does not include any of the interfaces from the top 10 of the most often implemented interfaces (and vice versa). In fact, the correlation between Interface Popularity and Interface Generality is only 0.1757 for all 829 interfaces of the JDK (and 0.2288 without `Serializable`): it appears that general interfaces are not very popular and vice versa.

4. CLASS STATISTICS

Like the interface metrics of Section 3, the class metrics presented next describe the relationship between classes and interfaces as expressed by the `implements` relationship, only this time from the classes' side. Since classes deliver the instances of a running system, the focus of these metrics is on the polymorphism of instances, i.e., their capability to take on different types.

4.1 JDK's most popular classes

In analogy to interfaces, we consider a class the more popular, the more references by clients it has. The formal definition for C*lass Popularity* (*CPOP*) is

$$CPOP_C = \sum_{i=1}^{n} \begin{cases} 1 \Leftrightarrow V_i : C \\ 0 \quad else \end{cases}$$

where V_i ranges over all variables in a project and $V_i : C$ means that the variable V_i is declared with class *C*. Since it is analogous to the Interface Popularity metric, Class Popularity can be expressed as a coupling fan in factor, too, only this time as the coupling fan in factor of the class (with coupling being defined as the number of variables declared with that class).

Table 3. The most popular classes of the JDK

#	Name of Class	CPOP
1	java.lang.String	16143
2	java.lang.Object	5684
3	java.awt.Component	1610
4	java.lang.Class	1342
5	javax.swing.JComponent	1077
6	java.awt.Rectangle	1006
7	java.awt.Dimension	997
8	java.awt.Color	900
9	org.omg.CORBA.TypeCode	749
10	java.awt.Graphics	704
11	java.util.Vector	635

Table 3 shows the results of applying Class Popularity to the JDK. Not surprisingly, `java.lang.String` is by far the most popular class. By contrast, its interface `CharSequence`, whose use should allow other implementations to take `String`'s place, is used only 95 times (Table 2).

The classes `java.lang.Object` (ranked 2nd), `java.awt.Component` (ranked 3rd) and `javax.swing.JComponent` (ranked 5th) are typical roots of (sub)hierarchies of classes. Other popular classes are container classes[6] (`java.awt.Rectangle` at rank 6, `java.awt.Dimension` at rank 7, `java.awt.Color` at rank 8 and `org.omg.CORBA.TypeCode` at rank 9).

At the 11th place we find the class `java.util.Vector`. Even though the direct use of `Vector` should be avoided, this class is referenced approx. 2.5 times more than its interface `List`, which contains most of the needed functionality and should be used instead. A possible explanation for this is the high ranking of `java.lang.Class` (ranked 4th), which is complemented by `java.lang.reflect.Method` at rank 31 (referenced 258 times). This is an indication of introspection being heavily used in the JDK and explains why interfaces such as `List` introduced in later versions did not generally replace their implementing classes. The subject is continued in Section 5.

4.2 Polymorphic Grade

Using classes in variables declared with interfaces increases the use of polymorphism in a program. The more interfaces a class implements, the more polymorphic situations can possibly arise in using the class, a property that has been termed *Polymorphic Grade* [2]. Note that Polmorphic Grade only gathers information about the supplier's view, not the client's view. It is defined as follows:

"[T]he polymorphic grade of a class [is defined] as the number of interfaces implemented by the class, independent of how much these interfaces overlap. This is to acknowledge that overlapping or even identical interfaces can nevertheless serve different purposes." [2].

More formally,

$$PG_c = \sum_{i=1}^{n} \begin{cases} 1 \Leftrightarrow C \le I_i \\ 0 \quad else \end{cases}$$

[6] Container classes are classes, whose instances' main purpose is to contain other objects or collections of other objects.

where I_i ranges over all interfaces in a project. Polymorphic Grade of a class is the class counterpart of Interface Generality. It is identical to the little known CLIFIN[7] metric defined in [10].

Table 4 shows the result of our analysis regarding the Polymorphic Grade of the JDK classes. The maximum number of interfaces implemented by a class is 20. The 4th column shows the results of the Polymorphic Use (PU) metric, which is explained in Section 4.4 below.

Table 4. The most polymorphic classes of the JDK

#	Name of Interface	PG	PU
1	java.awt.dnd.DnDEventMulticaster	20	1
2	java.awt.AWTEventMulticaster	18	0.996
3	java.beans.[…].BeanContextServicesSupport	12	1
4	com.sun.corba[…]vation.ServerManagerImpl	11	0.996
5	java.beans.beancontext.BeanContextSupport	11	1
6	javax.swing.JTable	10	0.927
7	com.sun.corba[…]._ServerManagerImplBase	10	1
8	com.sun.corba[…]icAny.DynValueBoxImpl	9	1
9	com.sun.corba[…]namicAny.DynValueImpl	9	1
10	org.omg.DynamicAny._DynValueStub	9	0.999
…			
170	java.util.Vector	5	0.456
...			
402	java.lang.String	3	0.017

The histogram of Figure 1 shows the concentration in the distribution of PG around zero and one implemented interfaces. Very few classes implement more than 8 interfaces. Interestingly, the top 10 of the most polymorphic classes contains mostly rather exotic classes.

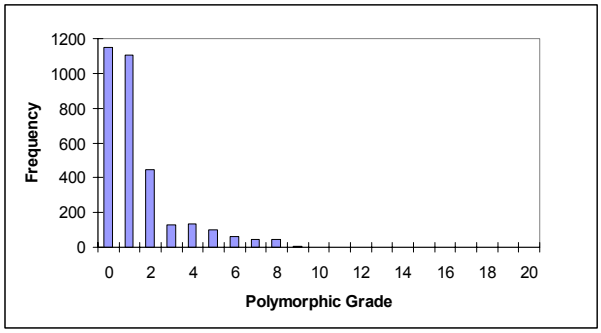

Figure 1: Histogram of the Polymorphic Grade

4.3 Versatility

Different interfaces need not specify different access protocols; they can overlap. The degree of that overlapping further qualifies the Polymorphic Grade of a class.

"Versatility measures the disjointness of the interfaces implemented by a class as

$$Vers_C = n - \frac{2}{n}\sum_{i=2}^{n}\sum_{j=1}^{i-1}\frac{|I_i \cap I_j|}{|I_i \cup I_j|}$$

where I_i stands for the set of features of the i^{th} interface of a class. Versatility values range from 0 (no interface implemented) to the polymorphic grade of the class (all interfaces pairwise disjoint). A versatility value of 1 means that all interfaces are identical. Higher values are indicative of the diversity of the class utilization — hence the name" [2].

Versatility cannot be expressed in fan factors.

Table 5. Versatility of classes in the JDK

#	Name of Interface	PG	Vers	PU
1	java.awt.dnd.DnDEventMulticaster	20	20	1
2	java.awt.AWTEventMulticaster	18	18	0.996
3	java.beans[…]ntextServicesSupport	12	11.5	1
4	java.bea[…].BeanContextSupport	11	10.7	1
5	javax.swing.JTable	10	10	0.927
6	com.sun.corba[…]rverManagerImpl	11	9.7	0.996
7	org.apache.xala[…]utputProperties	9	8.6	0.981
8	org.apache.xalan[…]StylesheetRoot	9	8.6	0.951
9	com.sun.corba[…]anagerImplBase	10	8.6	1
10	javax.swing.text.DefaultCaret	8	8	0.999
…				
195	java.util.Vector	5	4.76	0.456
…				
472	java.lang.String	3	3	0.017

The Versatility of `Vector` is 4.76. Its Polymorphic Grade being 5, this shows that there are some overlapping methods defined in the interfaces. The overlapping results from the similarity of the interfaces `List` and `Collection`: in fact, `List` extends `Collection` (and, for obscure reasons, re-declares all methods declared in the `Collection` interface).

4.4 Polymorphic Use

If a class implements interfaces, the instances of these classes can be accessed via an interface type. It is then interesting to know how often variable declarations make use of interfaces instead of using the class itself. In fact, this is what measures the degree to which the "program to an interface" directive is followed. The corresponding metric has been termed *Polymorphic Use* in [2]. In contrast to Polymorphic Grade, this metric assesses the client's view of a class and its implemented interfaces.

"The polymorphic use of a class relates the number of variables in a program typed with an interface implemented by the class to the total number of variable declarations assignment compatible with the class. A polymorphic use of 1 indicates that all instances of the class are accessed through interfaces, whereas one of 0 indicates that none are." [2]

Deviating from this definition, we define Polymorphic Use as follows:

$$PU_C = \frac{\sum_{i=1}^{n}\begin{cases}1 & \Leftrightarrow V_i : I \wedge C \le I \\ 0 & else\end{cases}}{\sum_{i=1}^{n}\begin{cases}1 & \Leftrightarrow V_i : I \wedge C \le I \vee V_i : C \\ 0 & else\end{cases}}$$

where V_i ranges over all variables in the project. Excluding superclasses from the denominator is to acknowledge for the fact that each superclass of C defines its own polymorphic use, which is not to be accumulated by its subclasses.

[7] CLIFIN stands for "class interface extension fan-in", which is another interpretation of the fan-in-metric. Another name would be the "inheritance fan in" of the class, if only interfaces are taken into consideration.

Polymorphic Use can be expressed as a combination of several fan factors. The numerator of the fraction defined above can be seen as the cumulated coupling fan in factors of the implemented interfaces of *C*, whereas the denominator may be seen as the cumulated coupling fan in factors of *C* itself as well as of the implemented interfaces of *C*.

Table 6. Polymorphic Use of the classes in the JDK

#	Name of Interface	PU	PG
1	java.awt.dnd.DnDEventMulticaster	1	20
2	java.beans[…]xt.BeanContextServicesSupport	1	12
3	java.beans[…]eancontext.BeanContextSupport	1	11
4	com.sun.corba[…]L._ServerManagerImplBase	1	10
5	com.sun[…].DynamicAny.DynValueBoxImpl	1	9
6	com.sun[…]ernal.DynamicAny.DynValueImpl	1	9
7	org.apache.xalan.templates.ElemApplyImport	1	8
8	org.apache.xalan.templates.ElemCallTemplate	1	8
9	org.apache.xalan.templates.ElemChoose	1	8
10	org.apache.xalan.templates.ElemComment	1	8
…			
261	java.util.AbstractSet	1	2
…			
1898	java.util.Vector	0.456	5
…			
1960	java.lang.String	0.017	1
1961	javax.swing.text.GlyphView	0	3

As can be seen from Tables 5 and 6, classes with high PG values also have high PU values. In fact, the correlation between PU and PG of the JDK's classes is 0.57, which should come as no surprise since the more interfaces a class implements, the higher is the probability that instances of this class are accessed over one of these interfaces. Correlations between PG and CPOP and between PU and CPOP are close to zero, however, indicating that these metrics are completely independent.

Interestingly, only 1003 out of the 3231 classes of the JDK (31%) are exclusively accessed through interfaces. The general-purpose class `java.util.Vector`, although implementing five interfaces and to a large extent covered by the `List` interface, has a polymorphic use of only 45.6%. The picture is much worse for the most popular class `java.lang.String`, which is accessed in only 1.7% of all cases through one of its interfaces, despite the introduction of its interface `CharSequence` in JDK 1.4.

5. INCREASING INTERFACE UTILIZATION

Given that the PU of all classes (and thus adherence to the "program to an interface" rule) is only 0.57 on average, one may wonder why this is so and what could be done to increase this value. We have chosen the `Vector` class from the JDK to perform some exemplary refactorings to assess what it takes to increase the PU value.

Ranked 11th in the hit list of the most popular classes, `Vector` is often accessed directly, even though it implements five interfaces (`Cloneable`, `Collection`, `List`, `RandomAccess`, `Serializable`).

Table 7, which shows the most frequently found access sets (with access set being defined as the set of methods called on a variable assignment compatible with the class; see [3]) of `Vector`, displays many legacy methods; the method most often used, `elementAt` (referenced 1080 times in the JDK), as well as `addElement` and others (see Appendix B) should really have been replaced by the corresponding methods from the `List` interface.

Table 7. Top 10 different access sets of `Vector`

#	Freq.	Access Set
1	100	elementAt(int):Object, size():int
2	61	addElement(Object):void, elementAt(int):Object, size():int
3	49	addElement(Object):void
4	34	add(Object):boolean
5	30	iterator():Iterator
6	21	get(int):Object, size():int
7	20	addElement(Object):void, elements():Enumeration
8	17	size():int
9	14	addElement(Object):void, copyInto(Object[]):void, size():int
10	10	iterator():Iterator, size():int
	10	addElement(Object):void, elementAt(int):Object, removeElementAt(int):void, size():int

To eliminate this legacy problem, we have first renamed `Vector`'s legacy methods to their equivalent pendants from `List`, by using the "Change method signature" refactoring. This alone does not change the polymorphic use of `Vector`, however, because the clients are still using variables of the `Vector` class.

In a second step we have therefore performed the refactoring "Use interface where possible" (which is part of the refactoring suite implemented with IntelliJ IDEA [10][8]) to substitute the references to the `Vector` class by references of the interface `List` wherever possible. Note that although this refactoring does not introduce static typing errors, correctness of the resulting code cannot be guaranteed, as long as methods are called using the introspective capabilities of JAVA. This however is difficult to decide; the heavy use of the `Class` class suggests that problems are likely to occur.

Table 8. Top 10 different access sets after the refactoring

#	Freq.	Access Set
1	121	get(int):Object, size():int
2	74	add(Object o): boolean
3	68	add(Object o):boolean, get(int):Object, size():int
4	30	iterator():Iterator
5	17	size():int
6	13	add(Object o):boolean, get(int):Object, remove(int):Object, size():int
7	12	add(Object o):boolean, contains(Object):Boolean
8	10	get(int):Object
9	10	iterator():Iterator, size():int
10	9	add(Object o), size():int

After this second refactoring, which changed the hit list of different access sets as shown in Table 8, we found that the Polymorphic Use of `Vector` has increased from 45.6% to 84.5%. At the same time, the Interface Popularity of `java.util.List` has grown from 234 (Table 2) to a value of 686, meaning

[8] This refactoring is also available in the ECLIPSE framework, where it is called "Use supertype where possible".

that after the refactoring `List` would be ranked 2nd in the IPOP top 10. Conversely, Class Popularity of `Vector` has decreased from 635 to 183.

6. CONCLUSION

In order to allow for a more thorough understanding of the goals of interface-based programming and the corresponding metrics suggested in [2], formal definitions have been provided and complemented with a comparison to the better-known fan factors [4]. The results of the application of these metrics on the JDK provide interesting insights into the usage of interfaces. In particular, we detected a certain dominance of so-called enabling [7] interfaces, comprising the tagging (or marker) and the Listener interfaces of the JDK (and possibly more). Also, while the general utilization of interfaces does not seem very high, there are some "hot spots" in the code where interfaces are introduced and used on a massive scale, as for example in the collections framework (as the results from popularity tests suggest). Last but not least — and not surprisingly —, it appears that some of the uses of interfaces are owed to JAVA's lack of multiple class inheritance.

Regarding classes, the metrics indicate that many root and/or container classes are among the most popular, indicating a high utilization of polymorphism. Metrics analyzing the `implements` relationship from the classes' side suggest that most classes implement only few interfaces. This is aggravated by the fact that in most cases (64%) the interfaces of a class are also overlapping.

On the other hand, even though only 20% of the total types available in the JDK are interfaces, 31% of the classes are exclusively accessed through interfaces. While this is encouraging, many key classes still show a low value of Polymorphic Use.

To test the possibilities of bettering interface utilization, we applied a sequence of existing refactorings on the `Vector` class. As could be demonstrated, the relevant metrics values, especially Polymorphic Use (which is an important indicator of client/server decoupling) can be significantly increased in only few steps. Other refactorings that foster the use of interfaces are made available through the FUJI project (see below).

Software products are highly complex entities offering many possibilities for quantitative analysis. The possibilities of expressing software in numbers are unlimited, and so is the number of available software metrics. To overcome the "yet another metric" trap, the Goal Question Metric (GQM) paradigm has been developed [11]. Using this approach, metrics are derived by defining goals which should be achieved in the development of a system, by then asking specific questions as to whether the goals have been achieved, and only then by providing metrics answering these questions. Elsewhere we have applied the GQM approach to justify the definitions of the metrics presented here; the interested reader is referred to [12] for a complete analysis.

The metric results presented in this paper were calculated using a plug-in for INTELLIJ IDEA [13]. This plug-in is part of our *Framework for the Use of Java Interfaces* (FUJI), which focuses on providing tools and guidelines for a better use of interfaces in large object-oriented projects. More information, including a download of the plug-in, is available at

http://www.kbs.uni-hannover.de/fuji/

REFERENCES

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

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

[3] Mayer, P. Analyzing the Use of Interfaces in Large OO Projects. Proceedings of OOPSLA 2003, Anaheim, USA.

[4] Henderson-Sellers, B. Object-Oriented Metrics: Measures of Complexity, Prentice Hall, 1996.

[5] Java™ 2 SDK, Standard Edition Documentation Version 1.4.1 http://java.sun.com/j2se/1.4.1/docs/index.html.

[6] Flanagan, D. Java in a Nutshell. O´Reilly & Associates, Inc., 1997.

[7] Steimann, F., Mayer, P. Patterns of interface-based programming. submitted to ICSE 2004.

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

[9] ORBIX 2000 Programmers Reference C++, IONA Technologies PLC, 2000.

[10] Patenaude, J.-F., Merlo, E., et al.. Extending Software Quality Assessment Techniques to Java Systems. Seventh International Workshop On Program Comprehension, Pittsburgh, Pennsylvania, 1999.

[11] Basili, V.R., Caldiera, G., et al. The Goal Question Metric Approach. Encyclopedia of Software Engineering, John Wiley & Sons, 1984.

[12] Mayer, P. Eine Metrik-Suite zur Analyse des Einsatzes von Interfaces in Java. Bachelor's thesis. Hannover, 2003.

[13] JetBrains IntelliJ IDEA. http://www.intellij.com/idea

APPENDIX

A The collections interfaces of the JDK

`java.util.Collection, java.util.Comparator, java.util.Enumeration, java.util.Iterator, java.util.List, java.util.ListIterator, java.util.Map, java.util.Map.Entry, java.util.RandomAccess, java.util.Set, java.util.SortedMap, java.util.SortedSet`

B Correspondence of methods

java.util.Vector	**java.util.List**
`public Object elementAt(int index)`	`public Object get(int index)`
`public void setElementAt( Object obj, int index)`	`public Object set(int index, Object element)`
`public void removeElementAt(int index)`	`public Object remove(int index)`
`public void insertElementAt( Object obj, int index)`	`public void add(int index, Object element)`
`public void addElement(Object obj)`	`public boolean add(Object o)`
`public boolean removeElement(Object obj)`	`public boolean remove(Object o)`
`public void removeAllElements()`	`public void clear()`