

Analyzing the Use of Interfaces in Large OO Projects

Philip Mayer

Information Systems Institute, Knowledge Based Systems, University of Hannover
Appelstraße 4
D-30167 Hannover, Germany
pm@pmayer.net

ABSTRACT

Using partial interfaces, i.e. interfaces that cover only a subset of the total set of published methods of a class, has several advantages, among them being an increase in understandability of the code and extended substitutability of classes in frameworks. However, analysis of large frameworks such as the Java API suggests that partial interfaces are only sparsely used. We believe that this is partly due to the fact that introducing and maintaining partial interfaces is perceived as tedious by programmers [5]. Therefore, we have created a metrics suite and tool support to assist the developer in using partial interfaces.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features – *abstract data types, frameworks, polymorphism.*

General Terms

Measurement, Design, Languages

Keywords

interfaces, partial interfaces, context-specific interfaces, roles, OO programming, Java, metrics, tool support, refactorings

1. INTRODUCTION

Although the use of interfaces in variable declarations is widely accepted as a prerequisite for writing decoupled code [3], it seems that interfaces are rarely used in practice. A recent study showed that on average only 1 out of 5 variables in various large Java projects declares an interface as its type [5]. Here, we go one step further and investigate to which extent the interfaces used (or just offered for use) in a project limit access to an object through a variable to those features of the object actually needed from within the variable's context (a so-called *context-specific* interface). The idea is that a high specificity of the interface would result in a better perception of a class's usage in the given context and in an increase in the number of classes being able to implement the interface. In general, partial interfaces tend to be less restrictive, allowing a greater decoupling of classes, resulting in a higher degree of plugability of a design.

Partial interfaces can occur in different forms. In Java, a dedicated interface construct (covering partial specifications) is part of the language. In languages such as C++, abstract classes with

no implemented features qualify as partial interfaces if only a subset of the features of the concrete class is specified.

Without tool support, analyzing the usage of classes in all contexts of a large software project and identifying suitable context-specific or partial interfaces is clearly unfeasible. Therefore, we have designed a small metrics suite measuring the use of classes and their interfaces in software projects, and provide tool support for deriving and analyzing the metrics' results. Identified partial interfaces can then be introduced and maintained using a set of refactoring tools [5] developed in a companion project.

2. A SMALL METRICS SUITE FOR ANALYZING INTERFACE USAGE

We used the goal-question-metrics approach [1] to define the goal behind the use of partial interfaces and to derive questions and the set of metrics for determining the achievement of our goal. Here, we focus on the metrics, of which there are four, namely NODAS, ACD, BCD, and IMI.

2.1 Definition

NODAS (Number Of Different Access Sets). Each variable declaration creates a new context with a specific type and specific method calls. The NODAS metric returns a count of pairwise disjoint access contexts of a class and its implemented interfaces. The value ranges from 0 to 2^n , where n is the number of published methods of the class.

ACD (Actual Context Distance). ACD is an assessment of the distance between the methods available to a context (via the used type) and the methods actually used in the context (concrete method calls). A value of 0 indicates a perfect match, that is, all available methods are being used, whereas a value close to 1 indicates that only few of the available methods are being used (indicative of a missing suitable partial interface).

BCD (Best Context Distance). The declared type of a variable needn't be the best-matching interface available. The BCD metric finds better matches among already defined and implemented interfaces, and uses those matches to calculate a "best possible" ACD. The value of BCD ranges between 0 and ACD.

IMI (Interface Minimization Indicator). Methods available in interfaces, but not used in any context lead to unnecessarily high ACD/BCD values. The IMI metric detects such superfluous methods by dividing the number of methods actually used in the project by the number of methods available in the interface. A value of 1 indicates that all methods are being used whereas one of 0 indicates that none are. The metric is cumulated over all interfaces of a class to allow a quick assessment of the situation.

Copyright is held by the author/owner(s).
OOPSLA '03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

2.2 Interpretation

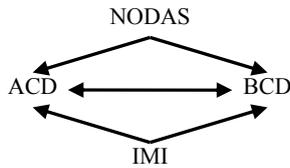


Figure 1: Interpreting the metrics

The first step in the metrics' evaluation is the comparison of the ACD and BCD metrics of a class. A large difference indicates that there are already interfaces available waiting to be used in the given contexts. The NODAS metric can help evaluate the absolute value of the ACD/BCD metrics, as a high number of different access sets justifies higher ACD/BCD values. So can the IMI metric, since a low number signals non-minimal interfaces which lead to unnecessarily high ACD/BCD values. Classes with high ACD/BCD values suggest missing partial interfaces – the *Context Analyzer* (see below) can then be used to determine what to do next.

3. RESULTS

Table 1 shows the top 5 classes of the Java API with respect to the NODAS metric. A comparison of the ACD/BCD values – they differ slightly – shows that not all contexts are typed with the best possible interface, yet the impact of replacing the types would be small. The IMI metric – being 1 – shows that none of the implemented interfaces has methods not used, so that minimizing the interfaces would be no remedy. However, the ACD/BCD values are close to 1 which suggests a lack of suitable partial interfaces. Having identified the problem, the *Context Analyzer* can be used to determine the missing interfaces.

Table 1: Top 5 NODAS classes of the Java API

Name	NODAS	ACD	BCD	IMI
java.lang.String	238	0.941	0.734	1
java.util.Vector	197	0.782	0.729	1
java.awt.Component	173	0.986	0.971	1
java.awt.Graphics	136	0.946	0.946	-
javax.swing.JComponent	116	0.978	0.968	1

Table 2, which has been produced by the *Context Analyzer*, shows the three most heavily used method subsets of the classes, the highest count being 568 usages of a subset containing only `equals`. The table also shows the types currently used in the contexts (C.Types) as well as the type best matching the context (B.Type) along with the distance to the given context (counted as the number of excessive methods).

In the case of `String`, a partial interface `CharSequence` including the methods `length` and `charAt` (and, per default, also `equals`) is readily available, but is only used in 2 out of 990 contexts; in the case of `Vector`, the interface `List` could have been used had the method calls been renamed as suggested by the Java 2 collections framework.

Of course, `Vector` and `String` serve as examples only; the real test would be application domain classes.

Table 2: Contexts for java.lang.String / java.lang.Vector

Count	C.Types	B.Type(dist.)	Methods
java.lang.String			
568	String	(any interface)	<code>equals</code>
279	String (277), CharSequence (2)	CharSequence (3)	<code>length</code>
143	String	CharSequence (2)	<code>length/charAt</code>
java.lang.Vector			
100	Vector	Vector (108)	<code>size/elementAt</code>
61	Vector	Vector (107)	<code>size/ elementAt/ addElement</code>
49	Vector	Vector (109)	<code>addElement</code>

4. TOOL SUPPORT

The metrics suite and the *Context Analyzer* have been implemented as plug-ins for IntelliJ IDEA [4] (an IDE offering a fully resolved parse tree). Together with the set of refactorings developed separately [5, 7], they make IDEA a very comfortable tool for the analysis, introduction and maintenance of partial interfaces. All described tools can be downloaded from [7].

5. OPEN ISSUES AND CONCLUSION

One problem not addressed here is that of context switches: If the content of a variable is passed to another context, the set of methods required from the object may change requiring typecasts to another interface. The implications of this circumstance prevent the introduction of minimal context-specific interfaces in certain cases, yet they do not hinder the use of partial interfaces in general, since related contexts together often form a role of an object, which is also naturally represented as a partial interface [2, 6]. A more thorough investigation of these implications is under way.

We have shown that our metrics and tool support can help identify possible partial interfaces. Further experiments are planned to evaluate the results of the metrics, and to gain new insights from applying the refactorings suggested by metrics and *Context Analyzer* to widely used packages.

6. REFERENCES

- [1] Basili, V., Caldiera, G., Rombach, D. The Goal Question Metric Approach. Encyclopedia of SE, Volume 1, 1994.
- [2] Fowler, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley 2000.
- [3] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Addison-Wesley 1995.
- [4] IntelliJ IDEA. www.intellij.com/idea
- [5] Steimann, F., Siberski, W., Kühne, T. Towards the Systematic Use of Interfaces in JAVA Programming. In Proc. of PPPJ 2003 (Kilkenny, Ireland, June 2003) 13–17.
- [6] Steimann, F. A radical revision of UML's role concept. In Proc. of UML 2000 (York, UK 2000) 194–209.
- [7] Framework for the Use of Java Interfaces. www.kbs.uni-hannover.de/fuji