Towards the Systematic Use of Interfaces in JAVA Programming

Friedrich Steimann
Institut für Informationssysteme
Wissensbasierte Systeme
Universität Hannover, Appelstraße 4
D-30167 Hannover
steimann@kbs.uni-hannover.de

Wolf Siberski
Learning Lab Lower Saxony
Universität Hannover
Expo Plaza 1
D-30539 Hannover
siberski@learninglab.de

Thomas Kühne
Fachgebiet Metamodellierung
Technische Universität Darmstadt
Wilhelminenstraße 7
D-64283 Darmstadt
kuehne@informatik.tu-darmstadt.de

ABSTRACT

JAVA's interface construct is widely perceived as a weak surrogate for multiple inheritance. Consequently, it should come as no surprise that despite their potential for writing highly decoupled code, interfaces are used rather sparingly. We have devised a conceptual framework for the utilization of interfaces in JAVA programs, and suggest tool support lessening the coding effort induced by the introduction and maintenance of additional interfaces, as well as a metrics suit measuring how and to which extent interfaces are actually used.

1. INTRODUCTION

It has been noted many times that variables (including instance variables and formal parameters) should be declared with interfaces, not classes as their types [1, 2]. This has the advantage that access to an object through a variable can explicitly be limited to those features of the object actually needed from within the accessing context, and that the actual class of the object is insignificant as long as it guarantees to implement the interface. Such is of particular importance in the development of frameworks, where a user's classes have to fit in at various plug points of the design, and in component-based programming.

The JAVA programming language comes with a type system that not only offers classes and interfaces as distinct syntactic concepts, but which also allows a single class to implement several otherwise unrelated interfaces. This offers the opportunity to use interfaces as *partial types*, i.e., as types that specify only one aspect (out of potentially many) of their implementing classes. Even though the type hierarchy of a program statically specifies which aspects each class may have, the set of aspects under which an object is being viewed at a certain point in time will change with the context in which it is being used, i.e., with the interface-typed variables that refer to it. Seen this way, an object can adopt different types in different contexts.

Despite this appealing property of JAVA's type system, it appears that in practice its interfaces are not used as one would expect. Indeed, the figures from Table 1 suggest that extensive use of interfaces is far from being reality: the ratio of class-typed to interface-typed variables in the JAVA packages we took a look at is 4:1 on average. In practice, it seems, the use of classes in variable declarations still clearly dominates over that of interfaces, begging the question why this is so. We speculate that it is partly due to the fact that introducing and maintaining interfaces in JAVA means considerable effort on part of the programmer, and partly due to the fact that an intuitive conceptualization of interfaces — comparable to that of classes — is still lacking.

2nd *Int. Conf. on the Principles and Practice of Programming in Java PPPJ 2003* (Kilkenny City, Ireland 2003) 13–17.
Copyright 2003 ACM 1-58113-000-0/00/0000…$5.00.

2. A USEFUL CONCEPTUALIZATION OF JAVA'S INTERFACES

It has been noted over and over that many application domain types are roles, not classes [4]. The omnipresent `Customer` for example, or `Student`, `Employee`, and `Manager`, are all roles which can be adopted and dropped by instances of class `Person`. That they can be adopted and dropped, that is, dynamically acquired and abandoned, is characteristic for roles. The perennial class `Person` on the other hand is somewhat more static: once a person, always a person. But not only dynamicity distinguishes roles from classes: whereas a class can stand alone, roles are invariably defined in the context of relationships. No `Customer` without a `Supplier`, no `Student` without a `University`, and so forth.

The concept of a role is quite old. Stemming from the world of theatre, it is used heavily today in disciplines as diverse as sociology and linguistics. Consistent with all uses of the term is that a role defines a certain behaviour or protocol demanded in a context, independently of how or by whom this behaviour is to be delivered. For instance, a role in a play describes text to be uttered and actions to be performed by an actor, but many other properties the actor may possess remain unspecified. A social role is associated with responsibilities towards and expectations by others, and any individual ready to cope with these should be able to fill the role. A semantic role is a logical position in a sentence set up by its predicate, which must be filled by a complement capable of assuming that role or function in the sentence, but whose nature can vary greatly. There appears to be consensus that a role specifies a certain protocol expected in a given context, without saying by what kind of entity it is to be delivered, or what else this entity should be capable of.

Table 1: Use of interfaces in widely used programming libraries and frameworks

	JDK 1.4	java.* only	javax.* only	org.jboss.*	org.eclipse.*
Types	5585 (100%)	1582(100%)	1867 (100%)	2584 (100%)	7584 (100%)
Classes	4736 (85%)	1340 (85%)	1662 (89%)	2035 (79%)	6554 (86%)
Interfaces	849 (15%)	242 (15%)	205 (11%)	549 (21%)	1030 (14%)
Ratio	5.6 : 1	5.5 : 1	8.1 : 1	3.7 : 1	6.4 : 1
Implements*	2906	702	1175	1666	3076
Implemented Interfaces/Class	0.6	0.5	0.7	0.8	0.5
Variables	88790 (100%)	31291 (100%)	28542 (100%)	30070 (100%)	127014 (100%)
Class typed	42930 (48%)	13416 (43%)	13914 (49%)	18905 (63%)	58597 (46%)
Interface typed	9510 (11%)	2061 (7%)	4043 (14%)	5094 (17%)	28857 (23%)
Ratio	4.5 : 1	6.5 : 1	3.4 : 1	3.7 : 1	2.0 : 1
Instance variables	10334 (100%)	3212 (100%)	3530 (100%)	5163 (100%)	19956 (100%)
Class typed	4747 (46%)	1293 (40%)	1654 (47%)	3036 (59%)	9851 (49%)
Interface typed	1030 (10%)	197 (6%)	496 (14%)	945 (18%)	3568 (18%)
Ratio	4.6 : 1	6.6 : 1	3.3 : 1	3.2 : 1	2.8 : 1
Parameters**	78456 (100%)	28079 (100%)	25012 (100%)	24907 (100%)	107058 (100%)
Class typed	31793 (49%)	12123 (43%)	12260 (49%)	15869 (64%)	48736 (46%)
Interface typed	8480 (11%)	1864 (7%)	3547 (14%)	4149 (17%)	25289 (24%)
Ratio	5.3 : 1	6.5 : 1	3.5 : 1	3.8 : 1	1.9 : 1

* sum of the number of interfaces implemented by each class

** return type counted as additional parameter

Roles, it seems, are partial specifications of objects; they share this property with JAVA's interfaces.[1]

2.1 Role vs. Class Types: An Ontological Distinction

Guarino has provided us with a crisp ontological distinction making a fairly unambiguous separation between roles and natural types [3]. This distinction may be paraphrased in object-oriented terms as follows:

– a type is a *role type* if
  1. for an object to belong to the extension of the type it must engage in a relationship associated with the type, and
  2. entering or leaving the extension of the type does not alter the object's identity;

– a type is a *natural type* if
  1. belonging to the type is independent of being engaged in a relationship (except for, perhaps, whole-part; see below) and
  2. an object cannot leave the extension of the type without losing its identity.[2]

Note how the definition of natural type nicely matches the class construct of mainstream object-oriented programming languages: the definitions of classes are outside the context of any relationships, and their instances keep their types for their lifetimes. In fact, the ontological foundation of the class concept may be seen as perfect justification for the lack of object migration in mainstream object-oriented programming languages. However, it does not explain the absence of a role construct.[3]

[1] As has been argued elsewhere, the roles of modelling can indeed be equated with interfaces in JAVA programming [5].

[2] Having said this, one should be aware that which concept is a class and which is a role depends on the specific universe of discourse. `User` for instance — a typical role — may be a class if it is the only representative of persons in a problem domain. Conversely, `Queue` — a typical class — can be represented as a role of `Vector` without problems (see below). Nevertheless, the ontological distinction remains valid; only the universe it is applied in changes.

[3] So, one might be tempted to blame the lack of role support in our OOPLs on the unavailability of object migration: if only an instance could change its type dynamically, it could enter the extension of role types as needed. This however misses one im-

2.2 Type Hierarchy and Inheritance

Roles and classes are organized in separate hierarchies. These hierarchies are interconnected by the `supports` relationship, defining the instances of which classes can play which roles. However, only class hierarchies are the natural realm of inheritance: the genes of a class (its implementation) are propagated from the general to the special. This regards not only the observables, but also and more characteristically the internals: inheritance is inherently genetic. Roles, on the other hand, make no assumptions about how a certain piece of functionality is achieved. Quite the reverse: roles explicitly leave it open instances of which natural types they are played by, allowing for genetically (i.e., in terms of inheritance) unrelated instances taking the same role. In a role hierarchy, only protocol is handed down from superrole to subrole.

2.3 Intension and Extension of Types

The intension of a class is its definition. The extension of a class is the set of its instances. While the intension of a class in JAVA is static, i.e., cannot change at runtime, its extension "breaths": it grows and shrinks in response to instance creation and disposal. It is therefore useful to distinguish between a static and a dynamic extension, the static extension of a class being the set of all instances of that class that can possibly exist, the dynamic extension being the set of instances existing at a certain moment in time (in a snapshot or state of the software system). Quite obviously, the dynamic extension of a class is always a subset of its static extension. As regards the class hierarchy, the extension of a class comprises the extensions of all its subclasses, both statically and dynamically.

Like that of a class, the intension of a role is its definition. However, unlike classes, roles have no instances of their own (they are abstract in a sense), they must recruit them from the classes supporting them. Like that of an abstract class, the static extension of a role is defined as the union of the static extensions of all the classes supporting it. This expresses that all instances of a supporting class are — in principle at least — capable of playing the role. The dynamic extension of a role, however, is usually only a subset of the dynamic extensions of the classes supporting it, namely the set of instances actually playing the role at that time.[4]

portant point: an object can play many different roles simultaneously, and this without ever giving up its natural type. But even if dynamic reclassification came in concert with multiple classification (allowing an object to be an instance of different classes at the same time), it would still lack the relationship aspect as one defining characteristic for the role concept. Instance migration is a concept for dynamic reclassification as a response to significant state changes; as will be seen, it is unnecessary to cater for roles.

[4] That the dynamic extension of a role is only a subset of the dynamic extensions of the supporting classes is often erroneously interpreted as indication that a role is a subtype of the role-playing class(es), which is wrong because statically they are not: all instances of a class supporting a role can, in principle, play the role, or they are not of the same type. That the dynamic extension of roles depends on the dynamic extensions of relationships is grounded in the ontological foundation of the concept, as detailed above.

Playing the role in this setting means that the instance is actually involved in a relationship, in that role. In programming terms being involved in a relationship means being assigned to a variable, be it an instance variable or a formal parameter. Transferred to JAVA programming, the dynamic extension of an interface is the set of instances assigned to variables typed by that interface.

3. THE PRAGMATICS OF USING INTERFACES

While we agree that interfaces should be used extensively in variable declarations, extensively does not mean exclusively. A good question to ask is therefore when to use interfaces and, equally important, when not to.

A simple test to decide whether or not to use an interface in the declaration of a variable is to refer to the conceptualization of interfaces, roles. Does the relationship represented by the variable give rise to the definition of a role? Does this role come with specific behaviour, or is it just a label that could be applied to any object? We shed light on a few typical cases.

Qualities Many variables hold qualities of an object, for instance `weight`, `height`, or `age`. Conceptually, the values (in the given example all numbers) play the "roles" `Weight`, `Height`, and `Age`, respectively. However, these roles are special in that the relationships they are defined by confine the natural type of the role players: they must be numbers. It is therefore not useful to introduce interfaces for qualities.

Aggregation Aggregation is a relationship with standard roles `Whole` and `Part`. However, it depends on the specific case what the whole requires from the part (or vice versa), so that predefined standard interfaces will be useless. If parts are private to and used only inside the whole, the definition of an individual part (or whole) interface makes sense only if it can be played by instances of different classes.[5]

Delegation The relationship between a delegator and its server is standard (with roles `Delegator` and `Recipient`), private (it is the secret of the delegator that is does not do the job itself), and, for most cases, does not require the introduction of special roles. For instance, when implementing a queue with the help of a vector, the relationship of `Queue` to `Vector` is the secret of `Queue`, the instance of class `Vector` held by an instance of class `Queue` is not meant to be replaced by the instance of another class (unless of course the underlying design decision changes) and thus an extra interface type is usually not justified.

Using library and API classes Library and API classes are typically multi-purpose, and a client will likely use them in contexts needing only parts of the offered protocol. For instance, in a certain context a `Vector` could be used as a `Queue`. In this case, it would be nice to have `Vector` support an interface `Queue`. However, libraries are usually outside the scope of development, i.e., they cannot be changed. Combined subclassing and interface implementation may be a remedy in these cases:

[5] Quite notably, ontologists have excluded whole-part from the set of relationships used to differentiate role from natural types, because many natural types are inherently defined as aggregates and thus defined in the context of a relationship, namely whole-part [3].

```
class PolymorphicVector extends
java.util.Vector implements Queue {

    static Queue newQueue() {
        return new PoylmorphicVector();
    }

    push(QueueElement aQueueElement) {
        add(aQueueElement);
    }
    …
}
```

4. A TECHNICAL FRAMEWORK SUPPORTING THE USE OF INTERFACES

To promote the use of interfaces in JAVA, we suggest a set of refactorings and other utilities that should ease the introduction and maintenance of interfaces during coding, and a set of metrics to measure the actual utilization of interfaces.

4.1 Refactorings and other Utilities

Build interface from used protocol New interfaces are often introduced on the fly, without prior knowledge of the set of features they should come with. In these cases, it is practical to first complete coding the context in which the (variable typed with that) interface is used and then to create the interface from the protocol used in that context. If one of the classes to implement the new interface is known in advance (and is already defined), automatic code completion can be made available by temporarily taking this class as the variable's type; it is later to be replaced by the interface extracted from the protocol used from that class.

Extend interface with available feature If a variable is declared with an interface missing a feature that its implementing classes possess, code completion should offer extended access to those features not included in the interface. Upon selection of one such feature, the interface definition should automatically be updated.

Extend interface with new feature If a feature not yet provided by its implementing classes is introduced to extend an interface, all these classes should be marked for completion.

Minimize interface Interfaces should be minimal, i.e., they should offer only the features actually needed in the contexts the interface is used. Because the same interface can be used in different contexts, it is not feasible to reduce the protocol of an interface to adapt to local needs alone. However, it is feasible to remove all features of an interface that are not used in any context in which the interface occurs.

Build interface hierarchy Interfaces can be arranged in a hierarchy, but this hierarchy need not be obvious at all times. Automatic analysis tools can be used to determine the overlaps of interfaces supported by the same (set of) classes and suggest the introduction of organizing superinterfaces and subinterfaces. This will be rather rare, though, and should be done only if it increases conceptual clarity and/or readability of a program.

Merge interfaces For all interfaces supported by the same (set of) classes, the merging of interfaces with same set of features together with the necessary renaming could be offered. Also, it may be reasonable to eliminate superinterfaces that have only one child and replace them with the subinterface. However, mergers that are based on accidental congruence of protocols rather than conceptual relatedness should be avoided.

4.2 Interface-Related Program Metrics

4.2.1 Class Metrics

Polymorphic grade We define the polymorphic grade of a class 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.

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

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

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.

Polymorphic use 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.

4.2.2 Interface Metrics

Generality 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.

Popularity 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.

4.2.3 System Metrics

System metrics should comprise those found in Table 1, average values of the class and interface metrics defined above, as well as histograms for distributions. The metrics from Table 1 are named and defined as follows:[6]

Interface to class ratio the total number of interfaces in a piece of code divided by the total number of classes; and

Interface typed to class typed variables ratio the total number of variables (instance variables, formal parameters, and temporaries) declared with an interface divided by the total number declared with a class.

[6] For better readability, the reciprocals have been used in the table. In the future, we hope to see higher utilization of interfaces, so that the ratios can be reversed.

5. RELATED WORK

The concept of role in conceptual and object-oriented modelling has been covered exhaustively in [4]; the technical correspondence of roles and interfaces is made plausible in [5]. Both works come with a comprehensive literature review to which the interested reader is referred. The ontological foundations of the role (and hence also the interface) concept are covered in [3]. The benefits of using interfaces are elaborated in countless papers; that they should be used to type variables is particularly put forward in [1, 2].

6. CONCLUSION

Although it is generally considered good practice to declare variables with interfaces, not classes, evidence we have collected suggests that this advice is not thoroughly followed by developers, not even for large and broadly (re)used code bases. We speculate that this disinclination is due to two major reasons:

1. the lack of a useful conceptualization of interfaces so that they are readily identified in the problem domain (and hence, like classes, have their natural places in a program design); and
2. the added effort entailed by the massive introduction of interfaces and their subsequent maintenance which, in today's coding environments, is still burdened on the programmer.

To better the situation, we have proposed a conceptual framework supporting the introduction of interfaces — thought of as roles — to JAVA programs, and supplemented it with possible tool support and a set of software metrics measuring the extent to which interfaces are actually used.

7. REFERENCES

[1] D'Souza, D.F., Wills, A. C. Objects, Components and Frameworks with UML (1998), Addison-Wesley, Reading, MA.

[2] Gamma, E. Helm, R. Johnson, R. Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software (1995), Addison-Wesley, New York.

[3] Guarino, N. "Concepts, attributes and arbitrary relations" Data & Knowledge Engineering 8 (1992), 249–261.

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

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