

What's in a Name?

Erik Ernst

Presented at the FICS'01 workshop

Abstract

Feature interaction may arise in many different ways, but one of the core topics is the issue of name binding: When two or more entities are composed, say A and B , and they provide more than one declaration of the same name, say n , should the composed entity contain one subentity under that name n , or should it contain several? If one, how should it be selected or constructed? If several, how could a client choose the appropriate one amongst them? This paper surveys the treatment of various problems in this area in several programming languages, thereby establishing a framework for the discussion.

1 Introduction

In many approaches to feature interaction management—including most other papers at this workshop—a software system is assumed to be constructed by certain means which are not of primary interest (the system could be written in some programming language), and the core topic is then to investigate this system by means of formal methods, testing, run-time checks, etc., in order to detect unwanted feature interactions, prove their absence, or similarly.

This paper complements all those techniques in that it focuses on the *construction* of software systems, on making it less likely that a software system is created with unexpected feature interactions in the first place. This is done by focusing on programming languages, and in particular by focusing on a mechanism which plays a crucial role in the emergence of feature interactions. That mechanism is *identification* of declarations, i.e., the rules that govern when two declarations will be considered as describing the same entity.

This implies that we do not consider many important kinds of feature interaction. E.g., one feature of an entity A could be some hard real-time guarantees, and one feature of B could be flexible, customizable support for logging; presumably an entity C created by composing A and B would not satisfy the same real-time requirements as A , because the logging in B could take unbounded time at points where A depends on almost immediate progress. Such kinds of interaction is beyond the scope of this paper. We concentrate on features associated with declared names in programs, so a real-time property would not be a feature. Moreover, feature interaction as considered in this paper will only occur when

two declarations are identified. However, we believe that even this restricted notion of feature and interaction includes a large class of important cases.

The topic of declaration identification is treated in Sect. 2. Section 3 deals with the *effects* of identification, i.e., the meaning of that single entity that is denoted by several declarations which are identified with each other. The last topic, considered in Sect. 4, is *problems* associated with those effects. Finally, Sect. 5 concludes.

2 Identification

The topic of declaration identification is the basis for feature interaction as it is treated in this paper. Given two different declarations of the same name n in some entity, those declarations are said to be *identified* with each other if the meaning of n in context of the entity as a whole must be described by considering the declarations as a group. Alternatively, the two declarations could have unrelated meanings, and there would then have to be some disambiguation rule in order to make both declared meanings of the name n available for clients—in that case the declarations are said to be *distinct*. For example:

```
class A { void f(); virtual void g(); };  
class B: public A { void f(); void g(); }
```

Box
1

In box 1, expressed in C++ [9], there are two classes A and B, and they both contain a declaration of two methods **f** and **g**. Since B inherits from A, an instance of B contains two method implementations called **f** and two method implementations called **g**. However, since **A::g** is declared to be virtual and **A::f** is not, the two declarations of **g** will be identified, and the two declarations of **f** will not.

We should note that identification is traditionally applied to declarations of methods but not to declarations of instance variables. This is probably because instance variables are mutable whereas methods are usually pure values, so instance variables may vary by simply assigning to them whereas methods can only vary if some other mechanism is employed. This makes methods less flexible than instance variables, but it saves a lot of space, it simplifies compile-time checking of the correctness of calling a given method on an instance of a given class, and it improves program comprehension, because the method implementation is known statically for each class.

2.1 Non-Identification

The oldest and simplest approach to declaration identification is to avoid it. In languages such as C [7] and Pascal [5] there are no mechanisms to compose entities such as Pascal **records** or C **structs** and thereby bring declarations together in the same context. Within one declaration block (e.g., the body of a **struct**), declared names must be distinct. Similarly, functions and procedures declared at the same level must have different names.

As a consequence, the question of identification never arises. Moreover, it is sufficient to investigate one declaration in order to learn about the declared entity—whatever holds

according to that declaration will always hold when accessing that entity. In languages with static bindings from name applications to name declarations, this enables excellent run-time performance of procedure calls and attribute access, and that probably made non-identification the default rule for member functions in C++ (like `f` in box 1).

2.2 Identification by Spelling

When identification by spelling is used, two declarations of a name n in one entity are identified with each other because the declared name is n in both cases. So, declarations of the same name in the same context are *always* assumed to denote the same thing.

At first, this rule might seem to be the one that is used in many dynamically typed languages, including Smalltalk [4] and Self [1]. After all, a message ‘`foo: x`’ sent to a Smalltalk object O gives rise to a lookup process (conceptually—it may be implemented in various ways) where the most specific class of O that provides a method with the name `foo:` is allowed to determine the meaning of `foo:` in context of that object.

In Self, a prototype based language that affords programmers an exceptional level of flexibility, the lookup rules are similar to the ones used in Smalltalk. One difference is that each object may have several parents (somewhat similar to having multiple superclasses in class-based languages). The rule applied here requires that there is only one most-specific declaration of the given name, otherwise an “Ambiguous method” error is raised.

It is still the case that the network of classes/objects is inspected, all declarations of `foo:` are considered as a group (conceptually, the implementation may optimize this), and one of them is selected for execution. However, in all these cases we might as well categorize the approach as belonging to the next group, because the number of arguments *are* taken into consideration when matching up method names, as explained in the next section.

2.3 Identification by Spelling and Argument Count

The syntax of languages like Smalltalk and Self ensure that the number of arguments is an integrated part of the name of a method—for instance, `foo` is a method that takes zero arguments, `foo:` takes one argument, and `foo:Bar:` takes two arguments, etc. A message send like `x foo: y Bar: z` invokes `foo:Bar:` on the object `x` with the arguments `y` and `z`. By using that syntactic form (supplemented with another form for ‘binary methods’) for all message sends, it becomes possible to determine the number of arguments expected by any method, just by looking at its name (its *selector*).

An interesting consequence is that while these languages lack static type checking, they will in fact never call a method with a wrong number of arguments—it would necessarily use a different method selector and thus designate a different method.

In these dynamically typed languages there may be several declarations of methods having the same name. In Smalltalk, there cannot be several instance variables having the same name in the same class, but in Self there is no distinction between instance variables and methods, so one object network may have several instance variables and/or methods with a given name. However, the effect of having multiple instance variables with the same

name is that the most specific one overrides the others; even though this can be used to simulate such features as class variables, it is also somewhat confusing and error-prone.

Finally, a `defmethod` form in a CLOS [6] program introduces a method implementation having a certain name and a certain argument list. This method will become one of the methods ‘contained in’ some generic function of the same name having a ‘congruent’ argument list (i.e., primarily, having the same number of required arguments), and when that generic function is called it will select some of the methods and invoke them. In this case the language indisputably identifies methods based on name *and* argument count, because CLOS does not encode the number of arguments into the method name.

Identification by spelling and argument count seems to work well for dynamically typed languages.

2.4 Identification by Signature

In several statically typed languages, including C++ and Java, declarations of methods with the same name may be considered distinct if they take a different number of arguments, or if the types of arguments are (sufficiently) different.

This means that we may have several methods named `foo`, and they are then partitioned into groups according to a combination of the name and the argument types. Within each group the declarations are identified, but declarations in different groups are not.

A common implementation technique used in C++ is to encode the types of arguments into the names of methods, resulting in so-called “mangled” names. Originally, this simplified linking of C++ programs using linkers created for C. Identification by signature may be thought of as identification by name, applied to the mangled names.

It should be noted that this technique of having many meanings of the same name in a given context makes it impossible to use that name in contexts other than the ones where the ambiguity can be resolved. For instance, it would not be possible to extend Java to allow usage of a method in any other way than calling it—e.g., we could not pass a method as an argument to another method—unless some disambiguation scheme were provided.

This approach is generally accepted in connection with statically typed languages even though it has some problems, as pointed out in the next section.

2.5 Identification by Identity

The last approach considered here uses an explicit syntactic representation of each group of identified declarations. It plays the role as the “identity” of that group, and the connection between any given declaration and the identity of the group that it is a member of is resolved at compile-time. Such an approach is taken in `gbeta` [2] and in LAVA [8]. For example:

```
A: (# v: < object #);  
B: A(# v: < integer #);
```

Box
2

In box 1, expressed in **gbeta**, there are two patterns (in this context, think of the word ‘pattern’ as a synonym for ‘class’) **A** and **B**, both containing a declaration of the name **v**. **B** is a subpattern (subclass) of **A**, so an instance of **B** will contain two declarations of **v**.

A declaration in **gbeta** may consist of a name (here **v**), then a colon, then a marker that indicates the kind of entity being declared (here ‘<’ or ‘:<’), and finally the right hand side that specifies the properties of the declared entity.

The marker ‘<’ in **A** indicates that this declares a pattern, it is virtual, and it is its own identity. The marker ‘:<’ in **B** indicates that this declares a virtual pattern. It is a requirement that every virtual pattern must have a statically known identity, so the declaration of **v** in **B** is only accepted by the static analysis because there is a statically known superpattern (namely **A**) that provides an identity for this virtual pattern (namely the ‘<’ declaration of **v**).

The word ‘virtual’ indicates that the declared entity is specified by means of a group of declarations, and it may be used for extensible methods (similar to virtual member functions in C++ and other OO languages), extensible classes (similar to type arguments for type parameterized classes and methods), and deferred objects (not described here, because of space constraints).

3 Effects of Identification

The most common effect of identification is *selection*, i.e., when the set of identified declarations has been determined, one of them is taken to be the active one and all the others are simply ignored. This is the case with methods in Smalltalk, all kinds of attributes in Self, virtual member functions in C++, and methods in Java. It provides us with the main stream OO semantics of “method overriding”.

For a CLOS method, all the member methods in a generic function are taken into consideration. The applicable ones (the ones whose argument constraints are satisfied by the actual arguments) are selected. Amongst the applicable methods, all the methods marked ‘before’ are executed first, then the most specific ordinary method, then all the methods marked ‘after’. Hence, this will *compose* an ‘effective’ method based on several declarations in the group of identified declarations.

In **gbeta** the approach is also to compose a resulting meaning out of the meanings of all identified declarations. This happens by means of C3-merging as described in [2, 3].

It should be noted that an approach capable of composing declarations seems to be more suited to work with separation of concerns than an approach based on selection, since the composition may bring together the separated concerns.

4 Problems Caused by Those Effects

Every one of these approaches can give rise to some problems. First, if two methods are conceptually unrelated, they should be able to coexist in one object without disturbing

each other. This works perfectly for non-virtual C++ member functions, but then we cannot redefine them. It is not supported in languages like Smalltalk, Self or CLOS, nor in Java (multiple interfaces with “the same” method must share the implementation of that method in every class implementing those interfaces together). It is supported in **gbeta**, in that two unrelated methods will have different identities and hence never be identified with each other. This means that a class in **gbeta** may have two separate methods with the same name *and* still support late binding of both methods. A special problem in this area is the confusion in C++ around identification of methods. For instance:

```

struct A { virtual void f() {}};
struct B: public A { virtual void f() {}};
struct X { virtual void f() {}};
struct Y: public X { virtual void f() {}};
struct BY: public B, public Y {};
struct BY2: public B, public Y { virtual void f () {}};
```

Box
3

There are three groups of declarations of **f** in this example. The groups are $\{A::f, B::f\}$, $\{X::f, Y::f\}$, and $\{BY2::f\}$. If we execute **f** on an instance of **B** or **BY** then we get **B::f**, both with **A** and **B** as the statically known class. Similarly, if we execute **f** on an instance of **Y** or **BY** then we get **Y::f**. This means that we do get late binding and that the first and second group of declarations above are considered distinct in **BY** that contains both. However, if we execute **f** on an instance of **BY2** then we get **BY2::f** in all cases.

This means that there are two distinct groups of declarations of **f** in all classes except **BY2**, where they are merged into one group. The problem with this is that a programmer who knows about the **A::f** “family” of declarations and who redefines it in **BY2** may accidentally capture the **X::f** family, thus destroying **X::f** for all callers. A programmer who does not know about the existence of **BY2** will have no hint that **A::f** and **X::f** may in certain cases be considered the same method. This problem does not exist when identification by identity is used, because the identities **A::f** and **X::f** would be different.

A similar problem is that a declaration of a method may accidentally override another method of the same name, e.g., if **B::f** were conceptually unrelated to **A::f** but the programmer overlooked the latter. With identification by identity it is syntactically marked out exactly which method declarations are supposed to introduce an entirely new declaration group and which ones are supposed to add a declaration to an existing (and statically known) group. This problem gets even worse in a language like CLOS, where the equivalent of `class BY` would already let **B::f** capture the **X::f** family, again silently.

In CLOS this can be used constructively in ‘mixin’ classes. However, **gbeta** demonstrates that mixins can also be used with identification by identity—a bit less flexible, but also safer.

Finally, it is a problem with very strict approaches like identification by identity that methods which are conceptually related but not declared to be so cannot be considered as the same method. However, since problems associated with missing identification generally show up at compile time, we think that it is less dangerous than problems associated with overly aggressive identification, which tend to be undetected until run-time.

5 Conclusion

In this paper, feature interaction was addressed with special attention to the topic of name binding, especially focusing on the subtopic of declaration identification and composition. Different approaches taken in various programming languages were presented, and a few problems associated with each of them mentioned. Generally, we take the position that declaration identification should be statically known, because unforeseen and conceptually unsound identification may cause disasters at run-time, whereas lack of identification that occurs as a compile-time problem for programmers may usually be handled by a bit of extra programming.

References

- [1] Ole Agesen, Lars Bak, Craig Chambers, , Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc., Mountain View, CA, 1995.
- [2] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.
- [3] Erik Ernst. Propagating class and method combination. In Rachid Guerraoui, editor, *Proceedings ECOOP'99*, LNCS 1628, pages 67–91, Lisboa, Portugal, June 1999. Springer-Verlag.
- [4] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA, USA, 1989.
- [5] K. Jensen and N. Wirth. *Pascal User Manual and Report*. Springer-Verlag, 1978.
- [6] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, USA, 1989.
- [7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
- [8] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings ECOOP'99*, LNCS 1628, Lisboa, Portugal, June 1999. Springer-Verlag.
- [9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.