# Object-oriented Encapsulation for Dynamically Typed Languages

Nathanael Schärli
Software Composition Group
University of Bern
Bern, Switzerland
schaerli@iam.unibe.ch

Andrew P. Black
OGI School of Science & Engineering
Oregon Health & Science University
Portland, Oregon, USA
black@cse.ogi.edu

Stéphane Ducasse
Software Composition Group
University of Bern
Bern, Switzerland
ducasse@iam.unibe.ch

## ABSTRACT

Encapsulation in object-oriented languages has traditionally been based on static type systems. As a consequence, dynamically-typed languages have only limited support for encapsulation. This is surprising, considering that encapsulation is one of the most fundamental and important concepts behind object-oriented programming and that it is essential for writing programs that are maintainable and reliable, and that remain robust as they evolve.

In this paper we describe the problems that are caused by insufficient encapsulation mechanisms and then present object-oriented encapsulation, a simple and uniform approach that solves these problems by bringing state of the art encapsulation features to dynamically typed languages. We provide a detailed discussion of our design rationales and compare them and their consequences to the encapsulation approaches used for statically typed languages. We also describe an implementation of object-oriented encapsulation in Smalltalk. Benchmarks of this implementation show that extensive use of object-oriented encapsulation results in a slowdown of less than 15 percent.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects; Inheritance*

## General Terms

Languages

## Keywords

Dynamic typing, Encapsulation, Encapsulation Policies, Information hiding, Smalltalk


*OOPSLA'04*, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.


## 1. INTRODUCTION

Encapsulation is widely acknowledged as being one of the cornerstones of object-oriented programming [22]. But what does the term mean? In a classic paper, Alan Snyder defined encapsulation as follows [28, p. 39]:

> Encapsulation is a technique for minimizing interdependencies among separately-written *modules* by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients, and thus between the designer of the module and other designers.

We have added the emphasis to point out that while this definition captures the essence of encapsulation for *modules*, it does not adequately define encapsulation in the context of *objects*. To see this, suppose that we are trying to encapsulate an object that maintains an internal data structure, such as a tree. We would like to protect the invariants of the tree, but clients need to traverse it. If we pass our users an *unrestricted* reference — an alias — to the tree, clients might modify the tree in a way that breaks the invariant. What we would like to do is to allow the encapsulating object full access to the tree, including the right to modify it, but to restrict the access that is granted to clients.

Solving this kind of encapsulation problem requires some sort of protection based not on modules but on individual references to objects. In contrast to *module encapsulation* as defined by Snyder, the primary purpose of these *object encapsulation* [8] techniques is not to facilitate code evolution, but to increase code reliability. It does this by allowing the programmer to implement data structures whose instances are guaranteed to be protected from the invocation of inappropriate operations on their subobjects.

Today, most statically typed object-oriented languages such as Java, C++, and C# provide relatively good support for module encapsulation, and many proposals have been made for augmenting the static type systems of such languages so that they can also express object encapsulation [2, 3, 8, 9, 13, 17, 20, 21, 25].

However, things are quite different in dynamically typed languages. Popular dynamically typed languages such as

Smalltalk, Self, Python, and Ruby still provide no encapsulation at all, or support it in a very limited way. Proposals such as MUST for an encapsulation model for Smalltalk [30] have never been adopted, either in Smalltalk or in any other popular dynamically typed language.

The encapsulation model [12] proposed by the developers of Self was rejected, and is not available in more recent versions of the Self language [1]. There have been proposals dating back at least as far as 1987 for extending Smalltalk with some (limited) features for object encapsulation [7], but 17 years later, such features are still not available in Smalltalk.

In this paper, we propose an object-oriented encapsulation model (OOE) for dynamically typed languages such as Smalltalk and Ruby. It provides a *uniform* and *expressive* mechanism to address most of the module and object encapsulation problems of which we are aware. OOE is compatible with dynamic languages because it is based entirely on message passing and has a simple semantics that makes it easy to understand and reason about.

OOE supports module encapsulation using Composable Encapsulation Policies [27]: the degree of encapsulation is not dictated by the implementor of a module, but is selected by the user subject to policies defined by the implementor. It also allows encapsulation policies to be associated with individual references to objects, which is sufficient to solve many, although not all, object encapsulation problems.

The rest of this paper is structured as follows. We first describe the problems that are caused by insufficient encapsulation, and develop a set of goals that an effective encapsulation mechanism should meet (Section 2). After giving a brief outline of our proposal for Object-oriented Encapsulation (OOE) in Section 3, we describe OOE in detail in Section 4, deferring to Section 5 the discussion of why we took particular design decisions, and the alternatives that we considered and rejected. In Section 6, we give a detailed description of our implementation of OOE in Smalltalk and use benchmarks to evaluate its impact on performance. In Section 7 we evaluate OOE against our goals; Section 8 describes related work and Section 9 concludes.

# 2. PROBLEMS AND GOALS

In this section, we set the context for our work, and motivate it by describing the problems that are caused by inadequate encapsulation mechanisms. Based on these problems, we then formulate a set of goals for an object-oriented encapsulation model for dynamically typed languages.

## 2.1 Encapsulation in Dynamic Languages

Why is it that encapsulation, which is such a well-established feature of statically typed languages, is so poorly supported in dynamically typed languages? We believe that there are three reasons.

First, most of the proposed encapsulation models address only a small subset of the various encapsulation problems, and so no one of them seemed to add enough value to justify the additional complexity. For example, the model proposed for Self [12] addresses the issue of how to prevent a method from being *called* from within another module[1], but does not address other module encapsulation problems such as preventing a method from being *overridden*, nor does it provide any help in the control of object aliasing. This model was in fact included in an experimental release of Self, but was soon removed because it was found to be too complex. An alternative proposal, from Noble, Clarke and Potter [24], suggests extending languages like Self with dynamic object encapsulation using techniques based on object ownership, but it does not address the fact that the base languages do not provide simple module encapsulation.

Secondly, many of the proposed encapsulation models are not well-suited to the lightweight, dynamic, and entirely message-based spirit of these languages. For example, the Smalltalk extension MUST [30] significantly affects the lightweight and dynamic character of the language by requiring the programmer to make quite *static* encapsulation decisions both when declaring methods and when sending messages. Other approaches, such as that of Noble, Clarke and Potter [24], are so restrictive that they would prohibit several common programming patterns such as external iterators [20] and are hard to implement efficiently (see Section 8).

Thirdly, language designers may have perceived that the kind of experimental programming for which dynamic languages were originally promoted — rapid prototyping and single programmer experimental projects — did not need encapsulation. This perception may have been compounded by the mismatch between available encapsulation techniques and the experimental style.

Agile programming methodologies such as extreme programming [6] make a virtue out of change and expand the reach of techniques previously thought suitable only for experimental programming to include customer-driven projects undertaken by a sizable team. The success of these methodologies has shown that the third reason was not well founded. In particular, communal code ownership demands that programmers be given a way of expressing their intent as to which methods are to be callable by which objects. Thus, we feel that there is a real need for encapsulation in dynamic languages, if only a mechanism can be found that is both adequate and appropriate. Indeed, the absence of static types makes the modularity improvements and reliability guarantees that come with powerful encapsulation mechanisms even more valuable in a dynamically-typed language than in one with a static type system.

## 2.2 The Problem of Interdependence

We agree with Snyder that one of the primary purposes of encapsulation is to minimize interdependencies among separately-written modules [28]. All object-oriented languages that are based on classes depend on the class as the fundamental unit of modularity. In this paper we focus on class-based languages: this means that our modules are classes. Hence, module encapsulation means class encapsulation, and we use the two terms interchangeably.

[1]Since the prototype-based language Self has no explicit modules, the developers instead suggested to use implicit modules consisting of an object together with its shared ancestors.

Two fundamental operations are defined on classes: inheritance and instantiation. Consequently, there are *three* ways in which classes depend on each other: (1) when they are in an inheritance relationship, (2) when one class instantiates another to create an instance, and (3) when that instance is eventually used. Although in many languages instantiation is a built-in operation of the language (for example, Java's **new**), in Smalltalk it is not: instantiation is accomplished by sending an ordinary message to the class itself. We will therefore not single-out instantiation in the discussion that follows; instantiation can be controlled in Smalltalk using exactly the same techniques as message send.

### 2.2.1 Interdependence through Inheritance

Most dynamically typed languages such as Smalltalk and Ruby do not allow a programmer to *hide* the internal implementation features (*i.e.*, methods and instance variables) of a superclass from its subclasses.

One aspect of this shortcoming is that the designer of a class cannot specify *access restrictions* that prevent some or all of its subclasses from accessing certain instance variables or calling certain methods. This has severe consequences for code evolution: whenever a feature of a superclass is modified, the programmer must check *all* its (direct and indirect) subclasses to ensure that the change does not break existing code. This is because any subclass might use the modified feature and may rely on its old meaning.

Another aspect of this shortcoming is that the programmer cannot specify that a certain feature of a class should be *statically bound*, that is, that all local references to the feature's name should always be bound to the local feature rather than to an overriding implementation that appears in a subclass. This is perhaps unsurprising in a language based on message passing and dynamic binding, but it too has serious consequences on code evolution. In fact, the consequences of this aspect are even worse. Not only must all subclasses be checked when an existing feature of a class is *changed*, but also all the subclasses *and superclasses* must be checked when a new feature is *added* [29].

To see this, imagine that a maintenance programmer detects some duplicated code in an existing class C and wants to extract it into a new internal method called check. To do this safely, the programmer has to make sure that the check method does not accidentally override an internal method with the same name implemented in any of C's superclasses. In addition, the programmer *also* needs to be sure that there is no method named check in any of C's subclasses, because such a method would override the new implementation of check that was intended to be internal to C.

This dependence on subclasses is particularly problematic: it is often impossible to check all the possible subclasses of a certain class, for example, because the programmer of the class works for a framework vendor and the subclasses are implemented by (and a trade secret of) its customers.

Python is one of the few dynamically typed languages of which we are aware that provides even limited support for decreasing the interdependence between classes that are related by inheritance. In Python, features whose names that start with two underscores are "private" in the sense that these names are valid only from within the class in which they are defined. Outside of that class, such features are available under a different name, which is derived from the original name by prefixing the class name.

Although this makes it unlikely that such an internal feature will be *accidentally* called or overridden outside of the class, it offers no real protection. Furthermore, the approach of protecting a method based on whether its name fits a convention is clumsy because it requires all the references to the method to be changed if the programmer decides to change the status of the method from "private" to "public".

### 2.2.2 Interdependence when using Instances

Whereas a class will typically be subclassed only a handful of times, it will be instantiated many times and its instances will be used from many other classes. Thus, it is even more important to protect the internal features of an instance from inappropriate access by a client than it is to protect them from a subclass. Unfortunately, most dynamically typed languages provide only very limited support for this sort of encapsulation.

In Smalltalk and Ruby, all instance variables are protected from direct access from outside the object that contains them. In contrast, all methods are externally accessible. Python is even less restrictive: by default, instance variables are fully accessible from the outside and there is no effective way of protecting them. Even if the programmer declares them as "private" by using a name starting with two underscores, they can still be accessed from the outside as described in Section 2.2.1.

None of these languages provide any support for declaring internal methods that cannot be invoked from outside of the class in which they are defined. This has severe consequences for code evolution: for every change to an existing method the programmer must check all the classes in the whole application to ensure that the change does not break existing code. This is because even if the changed method was *intended* to be reserved for internal use, there may still be invocations of this method from any other class.

Some Smalltalk dialects attempt to solve this problem by using a special naming convention to specify internal methods. In the Squeak dialect [19], for example, methods whose names begin with pvt are effectively private: the compiler ensures that these messages can be sent only to self. However, this approach not only prevents accesses to such internal methods from outside of their class but also from other objects of the *same* class. Thus, the pvt convention is a form of object encapsulation: in practice it is often too strict, because it prevents many commonly used data structures and patterns from being implemented. As with Python's double underscore, this approach is clumsy because changing the access attributes of a method requires renaming it.

The utility of the pvt feature is reflected in the Squeak image: although this feature has been available for years, only 9 out of about 40 000 methods in the latest Squeak image use it.

## 2.3 The Problem of Fragile Data Structures

Module encapsulation, as described by Snyder and implemented in most modern statically typed programming languages, minimizes the interdependencies between separately-written modules. However, it is not fine-grained enough to address the encapsulation problems related to object aliasing [18]. Reasoning about a class in an object-oriented program involves reasoning about the behavior of its instances, and those instances will depend for their correct operation on subobjects instantiated from other classes. If we do not have an object encapsulation mechanism that allows us to prohibit inappropriate manipulations of these subobjects (*e.g.*, through aliases), checking the correctness of a class may require reasoning about the whole program [8].

As an illustration, consider the class Morph, which is the root of the GUI framework in Squeak. Morphs have a hierarchic structure: a Morph contains an instance variable named submorphs, which is a (possibly empty) collection of Morph objects. Morph also implements a few methods such as addMorph:, removeMorph:, and moveMorphToFront:, which add and remove submorphs and change how they are layered. Since a Morph is responsible for properly displaying all of its contents, it must take some additional actions whenever its set of submorphs is changed, and it is therefore important that the Morph's clients always use these methods to modify the submorph collection, rather than doing so directly. With class-based encapsulation mechanism, the only way to ensure this is to make the reference to the submorphs secret and never pass it out of the parent morph.

Unfortunately, this conflicts with the need of some of Morph's clients to use the protocol provided in Collection to enumerate the submorphs. As a consequence, the implementor of Morph has to choose between one of the following unpleasant alternatives [20].

**Value semantics:** implement the method submorphs to return a *copy* of the submorphs collection. By avoiding aliases, this approach also avoids the problem of inappropriate manipulations through aliases. However, this is not a general solution to our problem because value semantics is not always appropriate and its use can therefore lead to subtle bugs. For example, if no special care is taken, it can happen that another thread adds or removes submorphs so that the copy returned by the method submorphs is out of date before the client actually used it. Another problem is that, depending on the usage scenario, this approach can require a large amount of unnecessary copying, thus incurring significant time and space penalties.

**Proxies:** instead of returning a reference to the submorphs collection itself, return a reference to a proxy [15] that serves as a protecting container for the submorphs collection. The proxy understands only a safe subset of the collections methods (*e.g.*, the enumeration protocol). In addition to being laborious to implement without language support, and introducing a forwarding overhead on every invocation, proxies have several methodological drawbacks.

1. To protect the real submorphs object consistently, the programmer must ensure that no reference to it is *ever* allowed to escape, either from Morph or from the proxy class. A single inadvertently leaked reference, whether through a parameter, return value or exception, defeats the whole scheme.
2. Whenever relevant methods are added, removed, renamed and changed in the class OrderedCollection, the proxy class may also need to be updated to ensure that all the safe messages are correctly forwarded and that unrestricted references to the submorphs object are not passed outside.
3. Subtle semantic problems can arise because of the different identities of the submorphs object and its proxy object.

**Fat interfaces:** instead of implementing a separate proxy class, one could implement all the necessary enumeration methods directly in class Morph. However, this is really just a variation of the proxy approach and it suffers from similar problems. It also increases the complexity of the already overly complex Morph interface. Most importantly, the standard names of the methods in the enumeration protocol cannot, in general, be used: whereas the submorphs collection understands do:, the parent Morph must instead implement methods like submorphsDo: and boundingPathDo: The need to use different message names destroys the uniformity of protocol that makes it possible to write polymorphic code, which is one of the major benefits of the object-oriented paradigm.

The implementation of Morph in Squeak currently uses value semantics: the method submorphs returns a copy of the submorph collection. However, to avoid excessive copying, Morph also provides some of the methods that would make up a fat interface: it implements the methods submorphsDo:, submorphsReverseDo:, submorphsIndexOf: and many others directly, although other enumeration methods such as submorphsCollect: are missing.

Morph is by no means unusual: there are many other well-documented examples that show the usefulness of object encapsulation for common data structures and patterns such as stacks and iterators [8, 23].

## 2.4 Goals

Our goal is to develop an encapsulation mechanism for dynamically typed languages that avoids the problems just described. We seek a mechanism that is expressive, simple, and appropriate for dynamic languages.

- *Expressive.* Our target mechanism must be expressive enough to solve the problems that we discussed in Sections 2.2 and 2.3. This means that it should facilitate code evolution by minimizing the interdependencies between different classes and that it should allow one to implement reliable data structures by controlling access to individual objects through aliases.
- *Simple.* It should add minimal complexity to dynamically typed languages. This means that the semantics of the language should remain simple and that the encapsulation mechanism should make it no harder to understand and reason about programs.

- *Appropriate.* The absence of type declarations makes programming in dynamically typed languages more lightweight than in their static counterparts. It also makes programming more experimental and incremental, for example, it is possible to execute and test a code fragment before all the type declarations are consistent or even present. Furthermore, the absence of static types allows classes to be reused in diverse and sometimes unanticipated ways. Our target encapsulation mechanism should support this dynamic style of programming: it should not burden the programmer with heavyweight type annotations, it should support an incremental style of programming and it should support flexible and unanticipated reuse.

# 3. OUR PROPOSAL IN A NUTSHELL

Object-oriented Encapsulation (OOE) combines the features of class and object encapsulation mechanisms. OOE defines all *variables* to be local, which means that they are completely hidden and inaccessible from outside of the structure in which they are defined.

The encapsulation mechanisms for *methods* are based on two cornerstones.

1. *OOE uses encapsulation policies [27] to specify the encapsulation properties of classes and objects in a uniform way.* Encapsulation policies can be shared among objects and classes. We allow different clients to access a given object or a class through different encapsulation policies; this is accomplished by associating encapsulation policies with *object references*.

2. *OOE defines encapsulation semantics by distinguishing between three different kinds of message send.* The distinction is *purely syntactic* and allows us to define a simple semantics that combines class and object encapsulation.

    - For *self-sends* and *super-sends*, distinguished by the keywords self and super, the receiver of the message is statically known to be the current object. Thus, object encapsulation is not relevant: only the encapsulation policies used in the inheritance chain of the receiver's class decide whether a message send is valid and how it is bound.
    - For *object-sends* (that is, all messages sent to object references other than self or super), the encapsulation policy that is associated with the target *object reference* is used to decide whether a message send is valid. In this case the target is treated as a black-box that is accessed through its *external* interface; internal details such as how the target's class is built from other classes are irrelevant.

# 4. OBJECT-ORIENTED ENCAPSULATION

In this section we explain in some detail our model for object-oriented encapsulation in dynamically typed languages. For concreteness, we do this in the context of Smalltalk. However, since our proposal relies on only the most fundamental features of a dynamically typed object-oriented language based on message passing, we are convinced that it could also be applied to other languages that fall into this category (*e.g.*, Ruby and Python).

For conciseness, this section presents our model without much discussion of our design decisions and without any attempt to justify them: this material is deferred to Section 5.

## 4.1 All Variables are Local

One purpose of our model is to control and reduce the dependencies between modules, which we assume to be class definitions. As a first step, we determine that instance variables are *never* visible outside of the class in which they are declared. This means that from within a class definition D, one cannot access, for reading or writing, any instance variables that are defined in another class C, even if C and D are related by inheritance. An immediate consequence of this rule is that the names of the instance variables in a class D can be chosen independently of the names of the instance variables in *all* the other classes. This provides stronger encapsulation than Smalltalk-80, which allows a method in a subclass to access the instance variables of a superclass.

As an additional restriction, we determine that each instance of a class can access only its own fields, and not those of other instances. This restriction is already present in the Smalltalk-80 language.

Besides instance variables, Smalltalk has the concept of *class variables* [16], which correspond to static fields in Java. As with instance variables, we would like to encapsulate class variables in the module in which they are declared. Therefore, we determine that a class variable that is defined in a class C can be accessed only from the class side of C: it cannot be accessed from the instance side of C nor can it be accessed from the class side of subclasses of C. The Smalltalk jargon "the class side of C" corresponds to the Java terminology "the static members of C". Thus, in Java terminology, our restriction says that static fields defined in a class C can be accessed only from the static methods of C, but not by non-static methods of C nor by static methods of subclasses of C.

Note that because access to instance variables and class variables can be granted through accessor methods (*i.e.*, getters and setters), these apparently severe encapsulation constraints for variables do not affect the kind of abstractions that a programmer can write. However, they do allow us to keep our model uniform and simple: we can now focus exclusively on the encapsulation of methods.

## 4.2 Using Encapsulation Policies

Every encapsulation model needs a way to specify what access rights should be associated with which methods. This is usually done by defining a set of keywords such as **public**, **private**, and **protected**, which can be used to annotate methods where they are defined. This provides fine-grained control over *what* can be accessed (individual methods), but very coarse-grained control over *whom* can perform the access (all code in one of a small number of pre-defined categories). One of the key benefits of our model is that we use *encapsulation policies* to specify access rights, and thus allow much more precision in controlling the *whom*.

The concept of encapsulation policies has been described and formalized in a language-independent way in a previous paper presented at ECOOP 2004 [27]. We will not repeat this materiel here, but will focus on the aspects that are relevant to Object-oriented Encapsulation. A summary of the relationship between this paper and the ECOOP 2004 paper can be found in Section 8.

#### 4.2.1 What is an Encapsulation Policy?

The basic idea behind encapsulation policies is to separate the encapsulation aspect of a class from the implementation aspect of a class. This separation allow these two aspects to be reused independently. The separation is accomplished by introducing a new entity, called an encapsulation policy, which is essentially a mapping from method selectors to access rights. Encapsulation policies are composable: this means that not only can new encapsulation policies be defined from scratch (*i.e.*, by explicitly specifying the access rights for each selector), they can also be defined by reusing existing policies, in a way that is independent of the inheritance hierarchy. Thus, encapsulation policies have a similar flavor to Java interfaces, which can also be defined in terms of other interfaces, and which can be reused across the inheritance hierarchy.

Encapsulation policies are used in two ways.

1. The *designer* of a class can associate an *arbitrary* number of encapsulation policies with the class. Each policy represents a set of encapsulation decisions that correspond to a certain pattern of use. It is important to note that these policies are independent of a particular mode of use (*e.g.*, inheritance or message send). However, the designer of the class can specify *default policies* for each mode of use; these apply if the client does not explicitly select another policy.
2. The *client* of a class can independently decide which encapsulation policy to apply and in what way the class will be used, *i.e.*, whether the class will be subclassed or whether its instances will be sent messages. The chosen policy then defines whether and how the methods defined by the class can be accessed (*e.g.*, whether they can be called or overridden). To avoid accessing the class in ways that were not intended by the designer, the chosen policy must be one that is provided by the class, or a restriction of one that is provided.

#### 4.2.2 Applying Policies to Individual References

The encapsulation model described in this paper extends the calculus of Composeable Encapsulation Policies [27] by allowing encapsulation policies to be applied to individual object references. Hence, each *reference* to a Smalltalk object has an associated encapsulation policy that defines the access to the object that is permitted through this reference.

This changes the way that Smalltalk objects are manipulated in several ways.

- *Instantiation.* Classes are always instantiated through an encapsulation policy (which may be the default policy for instantiation). However, this encapsulation policy is associated with the returned *reference* to the object, rather than with the object itself.
- *Message send.* In Smalltalk, the only thing that can ultimately be done with an object reference is to send it a message. It is the encapsulation policy associated with the object reference that determines whether or not the message send is valid.
- *Assignment.* When an object is assigned to a variable, passed as a argument or returned from a method, a new object reference is created. The new reference is a copy of the original and has the same encapsulation policy.
- *Restricting an object reference.* A programmer can, if the encapsulation policy of an object reference allows it, request an object reference with a more restricted encapsulation policy for a certain object. This is done by sending the binary "restrict" message | to the object. r | restrictingPolicy is a new reference to the same to which r refers; the associated encapsulation policy is the intersection of the policy of r and restrictingPolicy.
- *Obtaining object reference with a different encapsulation policy.* In certain cases, a programmer can also request a reference with a completely different encapsulation policy for a certain object. This is possible only if (a) the designer of the object's class explicitly allowed this by implementing an appropriate method, and (b) this method is accessible through the encapsulation policy of the available reference.
- *Object Identity.* The ordinary Smalltalk identity operator == compares object references without regard for the associated encapsulation policies: two object references are identical if they refer to the same object, even if the references have different encapsulation policies. However, since it is possible to retrieve the encapsulation policy associated with an object reference, a programmer can also check whether two object references have the same encapsulation policy.

### 4.3 The Varieties of Access Right

As was mentioned in Section 4.2.1, an encapsulation policy maps a message selector to a set of access rights. In this section we consider the kinds of access rights that can meaningfully be associated with a message.

Recall that we must consider access rights in two different circumstances: sending a message to an instance and inheriting from a class. The situation when using an instance is simple: either a message can be sent, or the message cannot be sent. This distinction is modeled with the call right **c**: a message m can be sent to a reference r if and only if the encapsulation policy associated with r maps m to a set containing **c**. However, note that a message that cannot be sent to reference r might still be sendable to the same target object through a different reference.

The situation is more complex when we consider inheritance. The ability for a class to declare a method that is "private", *i.e.*, neither callable nor overridable from its subclasses, is very important for maintainability. This is because a programmer can define such a method without having to check whether existing subclasses already define it (see Section 2.2.1). What, then, does it mean if one of the existing subclasses *does* define a method with the same name?

We say that this means that the method is *re-implemented*. It is *not* overridden, because sends of the private message in the superclass do not invoke the re-implementing method in the subclass. Instead, the subclass and the superclass each have their own method, and both are reachable from their own scopes. Note that the right to re-implement a method (which we denote by **r**) is orthogonal to the right to override it (denoted by **o**). It may be quite appropriate for a superclass to make a particular method re-implementable *and* overridable, and to allow one of its subclasses to override it while another re-implements it.

Thus, we see that for each method there are three independent access rights: **o**, **r** and **c**, and so there are potentially $2^3 = 8$ different sets of rights. Which of these combinations make sense? We determined (and discuss in Section 5.2) that in a dynamically typed language like Smalltalk, it is *not* reasonable for a superclass to prohibit re-implementation (although it may be quite reasonable for a subclass to decline the proffered **r** right). This leaves us with four sets of rights that a superclass can offer to a subclass for each of its methods: {**r**}, {**or**}, {**cr**} and {**cor**}.

{**r**} means "hidden": the method can be re-implemented, but not called or overridden.

{**or**} means "overridable": the method can be re-implemented or overridden at the discretion of the subclass, but not called.

{**cr**} means "callable": the method can be re-implemented or called, but not overridden.

{**cor**} means that the subclass has "full access".

Note that *none* of these sets of rights can be obtained in Smalltalk-80. This is because the language does not support the concept of re-implementation; a method implemented in a subclass always overrides the corresponding method in the superclass.

## 4.4 Controlling Access to Methods

Together with the access rights defined above, encapsulation policies give the programmer considerable flexibility to specify whether a method is callable or not, and whether it should override or reimplement a superclass method. In this section, we define how these encapsulation decisions affect the *semantics* of the language; *i.e.*, we define what it means for a method to be declared as not callable, and what it means to re-implement (rather than override) a superclass method.

Note that the exact form of these definitions is one of the most crucial issues in an encapsulation model for dynamically typed languages. Our choice is discussed and motivated in Section 5.1.

### 4.4.1 Three Kinds of Message Send

Smalltalk-80 distinguishes between *two* kinds of message send: normal sends and super-sends. Normal message sends are used to send a message to an arbitrary target object; they are represented syntactically by an expression denoting the target object followed by the message. Super-sends

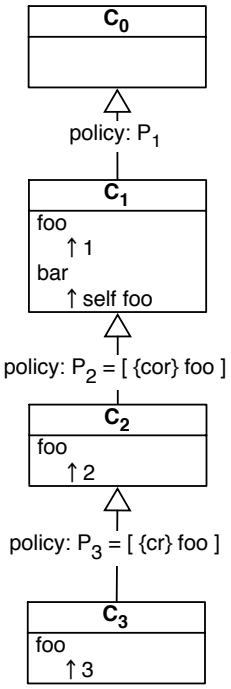


**Figure 1: A chain of subclass definitions showing encapsulation policies on the inheritance relation.**

differ from normal message sends in two ways: first, the target is always the current object (*i.e.*, self), and second they cause the message lookup to start, not in the class of the receiver, but in the superclass of the class that contains the super-send. Super-sends are *syntactically* distinguished by using the keyword super as the target.

In our encapsulation model we partition the set of normal sends into two categories with different semantics regarding encapsulation: *self-sends* and *object-sends*. As with super-sends in Smalltalk-80, the distinction is *purely syntactic*: a message send is defined to be a self-send if and only if the receiver of the message is the keyword self[2]. Object-sends are defined to be all the message sends that are neither self-sends nor super-sends. As a consequence, the expressions x foo and self foo have different semantics, even if the current value of variable x happens to be self: the first is always an object-send, while the second is a self-send.

### 4.4.2 The Semantics of Re-implementation

If a subclass inherits from its superclass using an encapsulation policy that does not allow it to override a method inherited from a superclass, the subclass is nevertheless allowed to re-implement the method. In other words, it is allowed to define what is conceptually a new method that happens to have the same name. This must necessarily affect the meaning of method lookup, but does so differently for the three different kinds of message send.

[2]If Smalltalk's cascade construct [16] is used to send multiple messages to the keyword self, all of the messages are self-sends.

We can now define the semantics of method lookup and re-implementation. Consider subclasses $C_0, C_1, \ldots, C_n$, where $C_0$ is the root of the class hierarchy and $C_i$ is a subclass of $C_{i-1}$, using the encapsulation policy $P_i$, for all $1 \leq i \leq n$ (see Figure 1). Consider a message m that is sent from an object of class $C_n$ by executing a method that is defined in class $C_k$, for some $k \leq n$. We define the method lookup for the three different kinds of send in terms of the ordinary Smalltalk message lookup algorithm, parameterized by the class where the lookup starts.

- *Object-sends.* If the message m is object-sent to an object $o$, the method lookup starts in the class of $o$.
- *Super-sends.* If the message m is super-sent, the method lookup starts in class $C_{k-1}$.
- *Self-sends.* If the message m is self-sent, we distinguish two cases.
  1. If there is a smallest index $i$ with $k < i \leq n$ such that $P_i$ declares the message m not to have the **o** right, then the method lookup starts in class $C_{i-1}$.
  2. If no such index $i$ exists, the method lookup starts in class $C_n$.

This means that for both object-sends and super-sends, the method lookup is the same as in Smalltalk-80 and entirely independent of encapsulation policies. Only self-sent messages are affected by the encapsulation policy.

As an illustration, consider Figure 1 and imagine that the message bar is sent to an instance of $C_3$, which means that $n = 3$ and $k = 1$. When the method bar (defined in $C_1$) executes self foo, which foo method is invoked? In the figure, $P_2$ does associate the **o** access right with foo, whereas $P_3$ does not. Hence $i = 3$ and the method lookup for foo starts in class $C_2$, where foo is indeed found and invoked.

### 4.4.3 Valid Message Sends

We can now also define the encapsulation restrictions that apply to message sends. We assume that our message send is located in a class $C$, which is defined as a subclass using the encapsulation policy $P$, and we then define *valid* message sends as follows.

- *Object-send.* An object-send of a message m to an object reference $r$ is valid if and only if m has right **c** in the encapsulation policy that is associated with $r$.
- *Super-sends.* A super-send of a message m is valid if and only if m has right **c** in $P$.
- *Self-sends.* We partition a self-send of a message m into two cases.
  1. If the method lookup of m yields a method that is implemented in $C$ or one of its subclasses, then the send is valid.
  2. If the method lookup of m yields a method that is implemented in one of $C$'s superclasses, then the send is valid if m has right **c** in $P$.

Thus, for self-sends and super-sends, the only relevant encapsulation policy is that of the class defining the method. In contrast, for object-sends, the only relevant encapsulation policy is that associated with the object reference. An attempt to send an invalid message generates a *messageInvalid* error. This is similar to, but distinct from, a *messageNotUnderstood* error, and can be handled by the programmer in a similar way.

## 4.5 Examples

To clarify these definitions, we now present some examples of how our encapsulation model can be used. We do this using our previous Smalltalk-based syntax and policy management proposal [27]. An encapsulation policy literal is a list of message selectors between brackets [ ]. The meaning of such a literal is that the listed message selectors are fully accessible, *i.e.*, they are mapped to the set {**cor**}, whereas message selectors that are not listed are hidden, *i.e.*, they are mapped to the set {**r**}.

### 4.5.1 Simple Class Extension

The first example defines Set as a subclass of Collection.

```
(Collection subclass: Set)
    instanceVariableNames: 'array tally';
    policyAt: basicUse put: [add: addAll: remove: ...];
    policyAt: basicExtend put: basicUse + [keyAt: scanFor: ...].
```

The new subclass offers its clients two encapsulation policies, named **basicUse** and **basicExtend**[3]. **basicUse** is defined using a literal: it associates full access with the message selectors add:, addAll:, remove:, *etc.* The policy **basicExtend** is composite: it grants all the access rights defined in the policy **basicUse**, but it also associates full access rights with the message selectors keyAt:, scanFor:, *etc.*.

Note that policies associated with the names **basicUse** and **basicExtend** are *default policies*: in the absence of an explicitly stated policy, **basicUse** is used as the default policy for new instances and **basicExtend** is used as the default policy for new subclasses. The default rule was used in the definition of class Set: since we did not explicitly specify a policy through which Set inherited from Collection, the **basicExtend** policy of class Collection was used. Similarly, if we create an instance of Set without explicitly specifying the encapsulation policy that should apply to the instance, the default policy **basicUse** defined in the class Set is used. However, we can also create instances with the less restrictive **basicExtend** policy by sending the message newWithPolicy: to the class Set.

```
x := Set new.  "uses default policy basicUse"
x add: 1.
y := Set newWithPolicy: basicExtend.
y add: 1.
y scanFor: 1.  "valid message send"
x scanFor: 1.  "error! invalid message send"
```

We can also use a more restricted interface to create a subclass of Set called RandomizedSet. RandomizedSet defines a new method randomDo:, which enumerates the elements in

[3]Those familiar with Smalltalk syntax should note that we use a **distinguished font** for symbols rather than a prefix #.

a random order and is implemented using a helper method generate that returns a random number. OOE lets us state explicitly that the class RandomizedSet is not intended to override any of the methods of the class Set. In the code below, this is done by specifying that RandomizedSet should inherit from Set through the encapsulation policy **basicUse** noOverride, which is the policy that allows the subclass to call all the methods in the policy **basicUse** offered by Set but disallows the overriding of any of them.

```
(Set subclass: RandomizedSet withPolicy: basicUse noOverride)
    instanceVariableNames: 'seed';
    policyAt: basicUse put: super + [randomDo:];
    policyAt: basicExtend put: super + [randomDo: generate]
```

This minimizes the interdependencies between RandomizedSet and its superclass, and has the advantage that even if someone later adds a new internal method generate to Set, it cannot be confused with the version in RandomizedSet. However, it should be noted that even though RandomizedSet uses a restricted encapsulation policy to inherit from Set, it is still possible for subclasses of RandomizedSet to use a more liberal encapsulation policy which will permit them to override methods in Set. Moreover, subclasses can inherit from RandomizedSet through any of the encapsulation policies that it offers. In our example, these policies are **basicUse** and **basicExtend**, which both grant full access rights to their message selectors and therefore allow the corresponding methods to be overridden. Note that both these policies are defined using the keyword super that refers to the policy of the same name in the superclass. (More details are available in reference 27.)

### 4.5.2 *Collection Hierarchy Example*

We now present a more elaborate example that shows how encapsulation policies can be used to create a simplified collection hierarchy. We first define the class Collection, which serves as the abstract root class of the hierarchy. This class is defined as a subclass of the class Object using its default policy for subclassing. The class Collection offers three encapsulation policies — **enumeration**, **readOnly**, and **basicExtend** — which allow the class to be used in different scenarios. This class does *not* offer a policy named **basicUse** because it is abstract and should not be instantiated.

```
(Object subclass: Collection)
    instanceVariableNames: '';
    policyAt: enumeration put: PCollEnumeration;
    policyAt: readOnly put: PCollReadOnly;
    policyAt: basicExtend put: PAll.
```

Unlike the Set example, the encapsulation policies used here are not literals but are globally named constant policies that are defined independently and can therefore be reused in many classes. The policy PCollEnumeration grants full access rights to the messages in the enumeration protocol, whereas the policy PCollReadOnly is built as the composition of the policy PCollEnumeration (which contains only observer methods) and a policy that contains some other read-only methods. The policy PAll is a special predefined policy that allows full access to all message selectors.

```
Policy named: PCollEnumeration
    is: [do: select: detect: collect: reject: ...]

Policy named: PCollReadOnly
    is: PCollEnumeration + [isEmpty notEmpty size ...]
```

Using Collection, we now define the subclass OrderedCollection (used to represent an extensible vector). Since the elements in OrderedCollection are sequenced, this class implements additional methods such as at: and first. To make these methods accessible, we add them into the encapsulation policies **enumeration** and **readOnly** that are offered by the class OrderedCollection. Additionally, because this class is concrete, we offer the named policy PCollSequenced (the definition of which is not shown) as **basicUse**.

```
(Collection subclass: OrderedCollection)
    instanceVariableNames: 'array firstIndex lastIndex';
    policyAt: enumeration put: super
                + [from:to:do: reverseDo: ...];
    policyAt: readOnly put: super + enumeration
                + [at: first last ... asReadOnly asEnumeration];
    policyAt: basicUse put: PCollSequenced;
    policyAt: basicExtend put: PAll.
```

These policies can now be used to create object references to instances of OrderedCollection under different encapsulation policies. This is done by implementing the methods asReadOnly, asEnumeration, and asSequencedCollection, which use the new language element selfWithPolicy: to return an object reference to the receiver of the currently executing method (*i.e.*, self) through the encapsulation policy that is specified by the argument. Note that, as for instantiation and subclassing, the argument to selfWithPolicy: must be an encapsulation policy that is offered by the current class (*e.g.*, **readOnly**) or a policy that is derived from such a policy using modifiers that make it more restrictive [27].

```
OrderedCollection>>asReadOnly
    ↑ selfWithPolicy: readOnly.

OrderedCollection>>asEnumeration
    ↑ selfWithPolicy: enumerate.

OrderedCollection>>asSequencedCollection
    ↑ selfWithPolicy: basicUse.
```

Given these definitions, we must prevent a user that has a **readOnly** or **enumerate** reference from sending it the message asSequencedCollection and thus acquiring a reference with a less restrictive encapsulation policy. This is done by making sure that the message asSequencedCollection is not declared to be callable in the encapsulation policies **readOnly** and **enumeration**. Similarly, asReadOnly must not be declared to be callable in the policy **enumeration**, because we intend that the access rights granted by **readOnly** are a superset of the access rights granted by **enumeration**.

Using this OrderedCollection implementation, we can now implement the method submorphs in the class Morph so that it returns an object reference that is restricted to the **enumeration** policy.

```
Morph>>submorphs
    ↑ submorphs asEnumeration
```

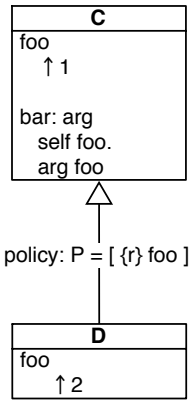


**Figure 2: The meaning of method hiding. C>>foo is hidden. Which of the sends of foo from method bar: invoke which of the foo methods?**

Similarly, we can add support for a fully protected read-only OrderedCollection by adding an instance creation method readOnlyWith: to the class side of OrderedCollection.

```
OrderedCollection class>>readOnlyWith: aCollection
    | inst |
    inst := self new.
    inst addAll: aCollection.
    ↑ inst asReadOnly.
```

This method takes a collection as an argument and then creates a new read-only collection that contains all the elements in the argument. Note that the object reference returned from this method is unique. It is the only reference to the newly created object, which can therefore never be mutated.

# 5. DESIGN DISCUSSION

In this section we review and discuss the main decisions in the design of OOE. First, we elaborate on the decision to base the meaning of method hiding on the distinction between self-sends and object-sends. Then, we justify our decision regarding the possible combinations of access rights, and finally we discuss how OOE fits into Smalltalk's "everything is an object" philosophy.

## 5.1 The Meaning of Method Hiding

As described in Section 4.3, a method with selector foo that is implemented in class C is "hidden" from a subclass D if a policy that assigns only the re-implement right **r** to the selector foo is used for the inheritance operation. To define *exactly* what such "hiding" means, we must answer the following two questions, which are illustrated in Figure 2.

1. If D also implements a method with selector foo, and a method bar: defined in C is invoked on an instance of D, *which* sends of foo inside bar: are *locally-bound* to the hidden method C>>foo rather than using the ordinary method lookup?
2. Which sends to the hidden method C>>foo are *valid*?

In statically typed languages, these questions are usually answered based on static types. However, in a dynamically typed language, no static types are available, and so OOE answers these questions by using a different semantics for object-sends on the one hand, and self- and super-sends on the other. This was one of the key design decision in OOE, and the reasons for this choice are not immediately obvious. In fact, we could have answered these questions based on three different kinds of information:

- the dynamic type (*i.e.*, the class) of the receiver,
- the identity of the receiver, and
- the different kinds of message send.

We now elaborate on the advantages and disadvantages of these three options. Because dynamically typed languages do not usually allow one to hide methods, we will first see how hiding is accomplished in statically typed languages such as Java, C++, and C#.

### 5.1.1 Based on Static Types

For concreteness, assume that the following Java methods are implemented in the classes C and D, and that c1 and c2 are two different instances of C, while d1 and d2 are two different instances of D.

```
class C {
    private int foo() {
        return 1;
    }
    public void bar(C arg) {
        C t;
        int i, j, k, l;
        t = this;
        i = this.foo();
        j = t.foo();
        k = arg.foo();
        l = new D().foo();
    }
}

class D extends C {
    public int foo() {
        return 2;
    }
}
```

In this code, the method C>>foo[4] is hidden by labeling its declaration with the keyword private. According to the Java semantics, this means:

1. A message send obj.foo is locally-bound to the hidden method C>>foo if the *static* type of obj is C.
2. A send to the hidden method C>>foo is valid only if it occurs within the definition of C.

[4]Even though we are talking about a statically typed langauge such as Java, we will nevertheless use the Smalltalk notation C>>foo to denote the method foo implemented in the class C.

As a consequence, when executing c1.bar(c2), the local variables i, j, and k get the value 1, because all the receivers of foo have the static type C, while the variable l gets the value 2 because the receiver new D() is of static type D. Executing d1.bar(d2) has the same effect: i, j, and k still get the value 1 because the three receivers this, t and arg still have the static type C, even though the objects to which they refer all have dynamic type D. Furthermore, the expression c1.foo() will not compile outside of the definition of class C, because the call to C>>foo is invalid.

The advantage of this approach is that it is *statically* observable which calls are locally-bound to the hidden method C>>foo and whether these calls are valid. Furthermore, the programmer can always decide which method should be called by choosing the right static type. However, this also means that as soon as a program contains methods that are declared as private, the static types are used not only to *check* whether a program is valid, but also to *determine* the meaning of the program by defining whether a method is late-bound! In other words, the types carry crucial semantic information.

### 5.1.2 Based on the Class of the Receiver

The above approach cannot be applied to dynamically typed languages because no static types are available. Instead, we considered defining the semantics of hiding based on the dynamic type (*i.e.*, the class) of the receiver together with some static information about the class hierarchy. We answered the two questions from Section 5.1 as follows.

1. A send of the message foo is locally-bound to the hidden method C>>foo if it occurs in the definition of C *and* the class of the receiver is C, D or a subclass of D.
2. A send to the hidden method C>>foo is valid only if it occurs within the definition of C.

At first glance, this definition seems appropriate because it guarantees that the method D>>foo is never called by a message send that occurs in the definition of C — in other words, it guarantees that all the sends of foo in C are "hardwired" to C>>foo, provided that the receiver is an instance of C, D or a subclass of D. Unfortunately, this is not always what the programmer expects. As an illustration, consider what happens if we translate the example from Section 5.1.1 into Smalltalk.

```
C>>foo
    ↑ 1

C>>bar: arg
    | i j k l t |
    t := self.
    i := self foo.
    j := t foo.
    k := arg foo.
    l := (D new) foo

C subclass: D withPolicy: [bar:].

D>>foo
    ↑ 2
```

The problem is that because there are no static types, the programmer has no way of specifying whether a send of foo should refer to the hidden method C>>foo or to D>>foo. Thus, it is not clear whether the programmer intended the expression (D new) foo on the last line of C>>bar: to refer to the method D>>foo as it does in the corresponding Java program or to the hidden method C>>foo. The same holds for the send of foo to the argument arg.

The difference between the type-based and class-based interpretations is reflected in the effect of executing the code c1 bar: c2. In the Smalltalk example, the local variable l gets the value 1 because the receiver is an instance of D and the send of foo is therefore locally-bound. However, in Java, the variable gets the value 2 as the static type of the receiver is D. Besides the fact that this may not be what the programmer intends and expects, the lack of explicit control makes it impossible to correct the situation.

Other problems with the class-based interpretation are that it makes static reasoning about the code much more difficult, and that is hard to implement efficiently. It is in general impossible to ascertain statically which sends of a hidden selector are locally-bound to the hidden method. Consider for example the expression arg foo in the method C>>bar:. This expression will be locally-bound to the hidden method if arg is of class C, D or a subclass of D. Otherwise, ordinary method lookup applies.

Implementation costs are especially high if frequently-used selectors are hidden in a large class hierarchy. As an example, suppose that the class RectangleMorph implements a new version of = that is incompatible with the semantics of = used in its superclass, Morph. RectangleMorph therefore inherits from Morph using a policy that hides the method =. However, this means that in all the dozens of places where = is sent in the classes Morph and Object (Object is the superclass of Morph), the implementation must check, at runtime, whether the receiver is an instance of class RectangleMorph or one of its 142 subclasses to know whether this send should be locally-bound to the hidden method.

This interpretation also causes severe semantic problems if it is applied in a language that supports multiple reuse of behavior through multiple inheritance or traits [26]. A problematic example is illustrated in Figure 3. Here we see a class C that, besides many other methods, defines methods foo and bar; method bar contains the expression self foo. Two classes D1 and D2 inherit from C and override the method foo, but do not implement any other methods. Now assume that new class E multiply inherits from both D1 and D2. It avoids a conflict between the two foo methods by hiding both of them, and then *re-implementing* a new method foo.

Clearly, this causes an ambiguity in the method C>>bar because it is not clear which of the two hidden foo methods D1>>foo and D2>>foo should be called by the expression self foo. This ambiguity can be resolved by overriding the message bar in E. However, all the other methods in C that send the message foo to an instance of E or one of E's subclasses are also ambiguous and thus also need to be overridden. Worse, it is not possible to ascertain statically which methods these are. As a consequence, *all* the methods that send

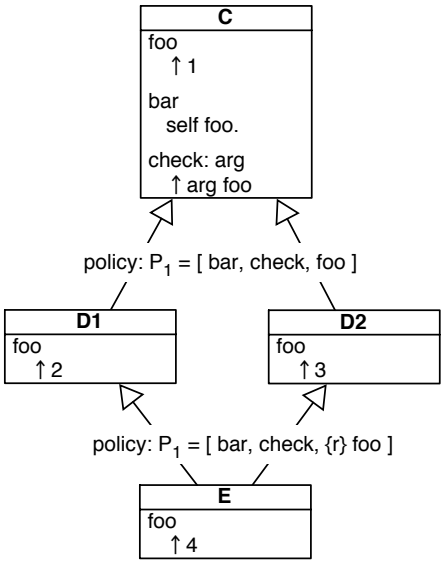


**Figure 3: a semantic problem with the class-based interpretation of method hiding.**

the selector foo would need to be overridden to ensure that there are no ambiguities at runtime. This may not be practical because it requires many methods to be overridden even if, in reality, they send foo only to instances of classes that are completely unrelated to E.

In our example in Figure 3, we would need to override the method C>>check: in E because it sends the message foo to the argument arg, which may or may not be an instance of E or one of its subclasses.

### 5.1.3 Based on the Identity of the Receiver

Another alternative is to define the meaning of hiding based on the *identity* of the receiver. This would mean the following.

1. A send of the message foo is locally-bound to the hidden method C>>foo if it occurs within the definition of C and the receiver is *identical* to the object that sends the message (*i.e.*, self).

2. A send to the hidden method C>>foo is valid only if it occurs within the definition of C and the receiver is identical to the object that sends the message (*i.e.*, self).

Compared to the previous approach, this alternative can be implemented somewhat more efficiently, because checking the receiver's identity is faster than checking whether the receiver's class directly or indirectly inherits from another class. Unfortunately, this check still cannot be done statically in most cases. As an illustration, consider again the method C>>bar:. Clearly, the message send self foo on the third line is always sent to the current receiver object and so it can be optimized statically. However, for all the other sends of foo, it is not immediately clear whether the message is sent to the current object.

A reasonable static analysis could be used to infer that t is bound to self on the second line and is not modified before the message foo is sent to it on the forth line. However, even using a sophisticated analysis, there will always be a large number of sends like arg foo on line five, where it is unlikely to be feasible to compute the identity of the receiver statically.

In addition to these implementation problems, this approach also leads to a semantics that may be hard to understand. As an example, compare the subtle semantic difference between executing the expressions d1 bar: d1 and d1 bar: d2. At first glance, one would expect these two pieces of code to have the same effect because neither C nor D has any instance-specific behavior. However, this is not the case: in the first expression the value of the local variable k is 1 because arg happens to be identical to self, whereas in the second expression, this value is 2 because arg and self are different.

Things get even more tricky if meta-functionality is involved. As an example, assume that the class C defines an instance variable counter with a setter method counter:, and a method C>>resetAll, which resets the value of counter for all the instances of C.

```
C>>resetAll
    self class allInstancesDo: [:each |
        each counter: each foo]
```

Executing d1 resetAll leads to the quite surprising result that d1 will have its counter value set to 1 whereas all the others instances of class D have their values set to 2.

### 5.1.4 Based on Different Message Sends (OOE)

Last but not least, we consider the interpretation that we selected for OOE, which is based on distinguishing self-sends from object-sends. This allows us to answer our questions in a way that is purely static.

1. A send of the selector foo is locally-bound to the hidden method C>>foo if it is a *self-send* that occurs within a method defined in C.

2. A send to the hidden method C>>foo is valid only if it is a *self-send* that occurs within the definition of C.

The advantage of this approach is that once we know that the class D inherits from C through an encapsulation policy that hides the method foo, we immediately know which message sends are locally-bound. For example, in the method C>>bar:, only the message send self foo on the third line will be locally-bound to the method C>>foo; ordinary message lookup applies to all the other message sends. This means that if we execute the code d1 bar: d2, the local variable i will get the value 1 while all the other variables j, k, l will get the value 2.

As a consequence, this approach can be implemented more efficiently than the alternatives that we have presented (see Section 6 for details about the implementation). Another advantage over the alternatives is its simplicity, which facilitates program understanding and reasoning about the code. Because the semantics have no dependencies on the identity of objects, the code in the classes C and D has no instance-specific behavior, and we can be sure that the expressions d1 bar: d1 and d1 bar: d2 are equivalent. Similarly, the avoidance of any dependance on dynamic type greatly simplifies things if OOE is used together with a mechanism that allows multiple reuse of behavior.

Programmers who have never programmed in a language where self-sends have a special semantics may at first find it confusing that even if self is identical to t (as in C>>bar:), the expression self foo is not necessarily equivalent to the expression t foo. However, our experience from analyzing and writing code in terms of this model has convinced us that this is actually a very natural concept, especially when used in the context of our object-oriented encapsulation model. We quickly began to think of a self-send as an "internal send", *i.e.*, a send that is issued from within an encapsulation boundary and which can therefore access hidden methods. In contrast, an object-send is an "external send", *i.e.*, a send that *always* accesses its receiver through the encapsulation policy associated with its reference, no matter whether the message was sent from within the receiver object or from within some other object that has the same class as the receiver.

Nevertheless, this approach does have some drawbacks. One of them is that treating self-sends and object-sends differently makes all message sends that have self as one of the arguments asymmetric. As an example, consider the two expressions self = arg and arg = self. One might expect = to be symmetric, but because one expression is a self-send and the other one an object-send, this may not be the case. Of course, this problem is not really new: message send is inherently asymmetric in all single-dispatch object-oriented languages, because only the receiver is taken into account when the message is dispatched. This issue is analyzed extensively by Bruce and his co-authors [11].

Another issue is that even if a method C>>foo is hidden from the subclass D, the class C may still contain sends of the message foo that finally call the method D>>foo. As a concrete example, assume that the method hasSameHash: is implemented in the class Integer and that the subclass LargeInteger re-implements the method hash.

```
Integer>>hasSameHash: arg
    ↑ self hash = arg hash
```

If this method is executed with instances of LargeInteger as both the argument and the receiver, the self-send self hash is locally-bound to the hidden method Integer>>hash whereas the object-send arg hash invokes the method in LargeInteger.

Unfortunately, such situations cannot be avoided with any of the alternatives considered in this paper. In fact, we would encounter the same behavior with the identity-based approach, while the approach based on dynamic types can lead to the opposite situation, which is that it calls the hidden method in cases where it should not. As we have pointed out above, this is a consequence of the fact that dynamically typed languages do not provide any mechanism that lets the programmer specify *explicitly* which method he means.

Being aware of this conceptual limitation, the approach of differentiating between self-sends and object-sends has the important advantage that the rule is extremely simple and purely syntactic. Thus, the semantics is clear based on inspection of the source code alone. In fact, once one is used to the fact that self-sends are conceptually different from object-sends, it is absolutely *not* surprising that in the above code, the two conceptually different sends of the message hash will call different methods if hash is hidden in a subclass.

Indeed, this is nothing more than a logical consequence of using Object-oriented Encapsulation: the self-send is an "internal send" to the object sending the message, whereas the object-send is an "external send" that treats the the object reference arg as a black-box and hence accesses it through the encapsulation policy that is associated with it. Thus, it is clear that even if the two receivers have the same class, the effects of the two sends may well be different because one send is seeing the features of the class from the inside whereas the other is seeing it through an encapsulation policy that might hide certain internal features.

On a more conceptual level, we have found that programming with OOE shifts the focus of the programmer's thinking away from classes, and towards objects and their encapsulation polices, *i.e.*, their *interfaces*. Whenever the programmer is dealing with an object, it is not really the class of the object that matters, but the interface that is associated with the reference to the object. This leads to a style of programming that emphasizes interfaces, indeed, programming against interfaces (rather than classes) becomes not only a recommended pattern but the only viable option.

## 5.2 Combinations of Access Rights

In Section 4.3, we determined that in a dynamically type language like Smalltalk, it is not reasonable to prohibit re-implementation of a method in a subclass. In the terminology of Java, this means that it is not possible to declare a method as final. While this decision may look quite controversial at first, it can be justified by looking at the motivations for declaring a method as final in Java.

According to Arnold and Gosling [4], there are two main motivations for final: security and performance. Declaring a method m in a class C as final can improve security because a programmer cannot declare a subclass that re-implements or overides m. Therefore, one can rely on its implementation wherever m is sent to an expression with static type C or a subtype of C. However, since Smalltalk is not statically typed, it offers no such type-based guarantees in the first place. As a consequence, declaring a method as final to improve security is pointless because another programmer can always replace an instance of C with an instance of a new class C′, which does not inherit from C and which can therefore provide an arbitrary method for m.

The performance benefit of declaring a method as final stems from the fact that it allows, in certain situations, the compiler to inline invocations of that method. Although this benefit applies equally to dynamically typed languages, we feel that including a language mechanism solely for performance is inappropriate in a higher-level language like Smalltalk. Not only does it make the language more complex, but it also tempts programmers to sacrifice extensibility of a class for the sake of a performance benefit that, in practice, is often negligible or irrelevant.

Neither are we convinced by the argument that final is appropriate for "perfect" methods that will never need to be redefined. This argument is especially unconvincing in the context of a dynamic language where code is often shared among multiple programmers. In our experience, it is hard if not impossible to know in advance how another programmer might want to use a class in the future.

## 5.3 Encapsulation Policies on the Class Side

In Smalltalk, classes are ordinary objects: they are singleton instance of so-called *metaclasses* [16]. This makes the language uniform, because all the concepts that apply to objects also apply to classes. The impact of this on encapsulation is that the same encapsulation mechanisms that we have defined for objects apply also to classes.

Because both class variables and class instance variables can only be accessed on the class side of the class where they are defined, a class must define accessor methods to permit its instances to access these variables. This raises the question of how such class-side accessor methods are encapsulated, *i.e.*, who is allowed to call them? Again, the answer is that because Smalltalk classes are objects, OOE's mechanisms for restricting access apply uniformly to methods defined on the class side.

On the class side, as on the instance side, the programmer can define encapsulation policies named **basicUse** and **basicExtend**, which are then used as the default policies for instances and subclasses. If there is a need to use a different policy, this can be achieved by using an extended subclass creation message such as subclass:withPolicy:classPolicy: that allows one to explicitly specify the encapsulation policy through which the implicitly created metaclass accesses its superclass.

One thing that is not possible is specifying a different instance creation policy for the class object itself: all class objects are created with the policy **basicUse**. This is a consequence of the fact that a class is a *singleton* object that is automatically created (using the default policy) when the class is defined.

In spite of all this uniformity, encapsulation policies have one special feature on the class side. In addition to the policies **basicUse** and **basicExtend**, class objects have a third default policy called **instance**. This policy is used whenever an instance accesses its class using the expression self class. This feature is important because it provides a way for a class to give its instances access through a different, and usually broader, encapsulation policy than that available to other objects. For example, this mechanism could be used to allow the instances of the class Delay to call the accessor methods of the class variable TimingSemaphore, while keeping that accessor hidden from other objects.

An examination of the Squeak image shows that in most cases it should be sufficient to have only the default encapsulation policies on the class side. Nevertheless, allowing custom encapsulation policies on the class-side is valuable as it will allow system engineers to hide the large number of internal meta-methods that are inherited from Behavior, ClassDescription, and Class. As an example, custom class encapsulation policies would make it possible to prevent programmers from inappropriately accessing the method dictionary using compiledMethodAt:.

# 6. IMPLEMENTATION

We have extended the Squeak language [19] so that it supports the encapsulation features presented in this paper. In this section, we first give a schematic overview of our implementation. Then, we we focus on some implementation details, and finally, we evaluate the costs of our implementation based on different benchmarks.

## 6.1 Schematic Overview

To implement OOE in Smalltalk, we needed to make sure that all message sends behave according to the semantics we have defined in Section 4.4. For self-sends and super-sends, this can be done statically, which means that that there is essentially no performance penalty compared to ordinary Smalltalk. For object-sends, this is not quite the case: for each send to an object reference that is protected by an encapsulation policy, we must ensure that the send is valid.

### 6.1.1 Changes to the Compilation Process

We have modified the compilation process so that it reflects the conceptual difference between self- and super-sends on the one hand and object-sends on the other hand. Whenever a method is compiled, we add it to the method dictionary *twice*, keyed by both the ordinary message selector and by an internal symbol that is distinct from all valid message selectors.

Whereas the ordinary message selectors are used for object-sends, the internal symbols are used for all self-sends and super-sends. This means that all the selectors that are self-sent and super-sent within a method are replaced by internal symbols. To ensure that the same internal symbol is used for all the occurrences of a given selector within a class, we store the mappings between the real selectors and the internal symbols in a per-class translation table. The first time that a selector occurs, the internal symbol is generated and stored in the translation table; it is then looked up on all subsequent occurrences of the same selector in the class.

Of course, self-sends and super-sends refer not only to methods that are implemented locally; they also refer to methods that are implemented in superclasses and subclasses. This means that during the compilation process, we have to make sure that the translation tables of classes that are related by inheritance map the same real selectors to the same internal symbols, unless the selector is not callable or is declared to be re-implemented by the encapsulation policy through

which a subclass inherits from its superclass. We ensure this by making sure that each translation appears in the table of only the topmost class that implements or calls the selector, and by using a lookup algorithm that is similar to the normal Smalltalk method lookup to do the translation.

The details of this algorithm are outside of the scope of this paper, but it is worth noting that it takes the access rights of the encapsulation policies used for the inheritance operation into account to make sure that all the self-sends are finally dispatched according to the semantics defined in section Section 4, and that invalid self-sends and super-sends are detected at compile-time.

### 6.1.2 Changes to the Virtual Machine

The compilation process just described ensures that all the self-sends and super-sends have the right semantics; no modification to the virtual machine is necessary and we can use the ordinary Smalltalk message lookup process. For object-sends, things are a bit more complicated. First, we need to model the fact that every object reference can have its own encapsulation policy. The simplest way to do this is to have each object pointer in the VM not point directly to an object but instead point to an *object reference*. Such a reference consists of two pointers: one to the object and the other to the encapsulation policy. Objects are represented as in the standard VM; encapsulation policies are represented as identity sets that contain all the callable selectors.

Finally, we must change the lookup mechanism for object-sends in the virtual machine. The new lookup process consists of two steps. First, we need to check whether the send is actually valid. This means that we have to check whether the selector is in the identity set that is associated with the object-reference. If it is not found, we raise a *messageInvalid* exception. Otherwise, the ordinary Smalltalk message lookup is applied and we proceed as usual: if the method is found in the method dictionary, it is executed; if not, we raise a *messageNotUnderstood* exception.

## 6.2 Implementation Details

In this section, we discuss some implementation details and point out the points in which our implementation differs from the schema described above.

### 6.2.1 Compilation Process

One difference between the conceptual schema described above and our actual implementation is that we do not actually replace all self-sends and super-sends with internal symbols. The reason for this is that typically only a small percentage of the methods are re-implemented in subclasses. Therefore, we can save space if we duplicate only the method dictionary entries for methods that are actually re-implemented in a subclass.

This means that when a new subclass is created and some methods are added, we at first neither replace any self- and super-sends with internal symbols nor do we associate the new methods with an internal symbol in the method dictionary. When a subclass actually re-implements such a method, we generate a new internal symbol for the original selector, add it to the translation table, create the necessary entries in the method dictionaries, and perform the necessary recompilations so that the self- and super-sends to the original selector refer to the newly created internal symbol instead.

The downside of this scheme is that it requires more work at compile time, especially if the programmer performs operations such as changing the superclass of a class. Whether this scheme is beneficial depends on the target platform (*e.g.*, whether memory is a critical resource) and on how frequently re-implementation actually occurs.

Our compilation process generates a special byte-code for self-sends. This is important because in the case of self- and super-sends, the virtual machine must not check whether the sent selector is in the encapsulation policy associated with the receiver. Since the Smalltalk byte-code set contains a few dozens of different send byte-codes, introducing a special "self-send version" for each of them was not an option. Instead, we introduced a single new byte-code that sets a virtual machine flag saying that the next send will be a self-send, and we modified the compiler so that it generates this byte-code immediately before each self-send.

### 6.2.2 Representing Object References

With respect to the virtual machine, the most important question was how to represent the object references. For simplicity, we decided to make object-references instances of a new class ObjectReference that contains two instance variables to contain the actual object and the relevant part of the associated encapsulation policy, which is an identity set containing the callable selectors.

The advantage of this representation is that each object reference is an ordinary object and can therefore be kept in the object memory without changing it. In particular, no changes to the garbage collection algorithm were necessary. The disadvantage is that this is not the most efficient representation. However, thanks to a few relatively simple optimizations we could bypass the most critical performance bottlenecks.

The first optimization is that we made ObjectReference a *compact class*, which means that the object header of its instances consists of only a single 32-bit word and contains the index of its class in the compact classes array. This makes object references small, and more importantly, it allows the virtual machine to check whether an object is an *object reference* or a *real object* by looking at the object header alone. The efficiency of this check is especially important because we do not represent every object as an object reference. One reason for this is that there are objects such as integers, floats, points, arrays, and strings that are instantiated very frequently, but usually use the default policy for instantiation (*i.e.*, the policy associated with the name basicUse).

Our implementation addresses this issue by allowing the programmer to *integrate* the default encapsulation policy into the method dictionary. To do this, we extended the representation of classes so that each class has *two* method dictionary pointers that by default both point to the ordinary method dictionary. However, if the programmer tells the class to integrate the default encapsulation policy, the

second pointer points to the *integrated method dictionary*, which is a method dictionary that maps *all* the selectors (including superclass selectors) that are declared as callable by the default policy to the corresponding method.

This has the advantage that all the instances of these frequently used classes do not need to be wrapped by an instance of ObjectReference as long as they use the default encapsulation policy. Besides the fact that this avoids the overhead for creating the wrapper, it also avoids any additional overhead for message sending. This is because we changed the virtual machine so that it uses the pointer to the integrated dictionary for all the object-sends to unwrapped objects, whereas it uses the pointer to the ordinary dictionary for all the self- and super-sends as well as for object-sends to wrapped objects. Since classes by default associate both of these pointers with the ordinary dictionary, this change to the virtual machine does not affect classes that do not integrate their default policy.

The downside of this optimization is that integrating all the callable selectors into a method dictionary uses more memory and requires additional care to keep the dictionary entries consistent when the default policy and the class are modified. However, as our current implementation uses this optimization for only a dozen or so classes (less than 1% of all the classes in the Squeak 3.7 image), the additional memory consumption is negligible.

#### 6.2.3 Method Lookup and Execution

As we have explained above, our implementation creates virtually no overhead to the message lookup in case of self- and super-sends. The same holds for object-sends that are sent to instances that use the integrated default policy. However, in case of object-sends to objects with an arbitrary encapsulation policy, the virtual machine has to perform a lookup in both the identity set representing the encapsulation policy and in the method dictionary.

To reduce the overhead associated with these two lookups, we extended the method lookup cache so that it takes into account not only the selector and the class of the receiver, but also the encapsulation policy. If an entry is found that matches all three of these parameters, we are done and can immediately execute the associated method without performing a real lookup at all. If the found entry matches only two parameters, we have to do a lookup in either the identity set representing the encapsulation policy or in the method dictionary. Only in the worst case when there is no match at all do we actually perform both lookups.

If the method lookup yields an ordinary (non-primitive) method, we can switch context and execute the method as usual, no matter whether the receiver and the arguments on the stack are object references or real objects. However, in the case of primitives, we need to be more careful. This is because it may or may not be necessary to unwrap the receiver and the argument before proceeding. The same holds for byte-codes.

As an example, consider the primitive at:put:, which is used to insert an object into an array at a certain index. When this primitive is executed, the receiver, the index and the object to be inserted are on the stack. Since the primitive expects the receiver to have the format of an array, it needs to be unwrapped. The same holds for the index, which needs to be converted into a C-integer. However, the object to be inserted must *not* be unwrapped. Instead, it must be inserted as a wrapped object reference to ensure that the real object does not "leak out" when the programmer retrieves the reference from the array.

| Benchmark | Orig. image | OOE image |
|---|---|---|
| Tiny benchmark (byte-codes) | 13.0 | 13.0 |
| Tiny benchmark (sends) | 10.0 | 15.0 |
| STones80 (low-level) | 12.6 | 12.9 |
| STones80 (medium-level) | 12.7 | 17.7 |
| Collection benchmark | 13.0 | 17.3 |
| Squeak MacroBenchmark | 10.4 | 13.7 |
| Average | 12.0 | 14.9 |

**Table 1: Performance overhead (in percent) of the modified Squeak virtual machine running an original image and an image that extensively uses OOE.**

### 6.3 Costs

In this section we evaluate the costs of our implementation in terms of execution speed and memory consumption.

#### 6.3.1 Execution Speed

To evaluate the runtime overhead of our implementation, we have compared the performance of the modified Squeak virtual machine with the original virtual machine by means of 6 benchmarks[5]. As a reference, we first executed the benchmarks in a Squeak 3.7 image on top of a copy of the original Squeak virtual machine (version 3.6.2), which we compiled using *gcc* 2.95.2. Then, we executed the same benchmark in the same image using our modified virtual machine, which had been compiled under identical conditions. Finally, we modified the image so that it made extensive use of encapsulation policies, and then again executed the benchmarks on top of the modified virtual machine.

Table 1 shows the results of this comparison. The numbers in the second column show the slowdown (as a percentage) that we encountered when we ran the benchmarks in the original Squeak image on our modified virtual machine. Since this image does not associate encapsulation policies with any instances (so wrapper objects are used), these values are a good indication of the performance penalty that is caused by our virtual machine modifications. In contrast, the numbers in the third column show the slowdown that we encountered when we ran the benchmarks in an image that applies an encapsulation policy to virtually all the objects involved in the benchmarks. This means that the overhead includes the time that is necessary to create all the necessary ObjectReference instances, to wrap and unwrap the real objects, and to check whether the object-sends are actually valid.

The first two rows show the results of two micro-benchmarks that are used to get a raw idea of how many bytecodes

[5]All the benchmarks were executed under Windows XP on a notebook equipped with a 1.2 GHz Pentium-III Mobile Processor and 512 MB RAM.

and message sends the virtual machine can execute per second[6]. STones80 is a benchmark that is available for many Smalltalk-80 dialects. Whereas the low-level version mainly involves arithmetic operations, array operations, block operations and object creation operations, the medium-level version also performs recursive calls, collection and stream operations. The collection benchmark is specifically written for the purpose of evaluating the performance of the OOE virtual machine, and it makes heavy use of nested collections. The result in the third column was measured in an image where every collection instance is wrapped. The only exception are instances of the class Array, which integrates the default encapsulation policy (see Section 6.2.2).

The last benchmark is a Squeak macro-benchmark that measures the time for opening, moving, resizing, and closing Smalltalk browsers on the screen. The result in the third column was measured in an image where most of the involved objects such as every morph (the GUI objects of Squeak), every collection, and every browser is wrapped. Again, we used the integrated default policies for basic classes such as Integer, Strings, Array, Point, and Rectangle.

As an overall result, these benchmarks show that the total performance overhead for extensive use of OOE in Squeak is about 15%. Given that we implemented our prototype from scratch in just a few days, that we have not yet done any performance tuning, and that we did not have any previous experience with the Squeak virtual machine (or with any other Smalltalk VM), this seems to be an acceptable result. Nevertheless, we believe that there is still a large potential for improvements. One indication for this is the fact that the performance penalty of our modified virtual machine is about 12% even if we do not use any encapsulation policies and wrapper objects at all. This means that the majority of the overhead is caused by checking whether an object is wrapped and maintaining the extended method lookup cache, whereas only a small amount of the time actually goes into the additional method lookup, the creation of the reference objects and the wrapping/unwrapping.

#### 6.3.2 Memory Consumption

Although our implementation strategy does not require the duplication of any source code or byte code, it does increase memory consumption for the following reasons.

- For each re-implemented method, an internal symbol has to be maintained in the translation table, and an additional association objects is inserted into the method dictionaries.
- Each class has an additional field to contain the integrated method dictionary. However, less than 1% of the classes actually maintain such a method dictionary; in the vast majority of classes this field points to the ordinary method dictionary.
- An identity dictionary containing all the callable selectors needs to be maintained for each encapsulation policy that is associated with an object-reference.
- At runtime, an instance of ObjectReference is used whenever a (non-integrated) encapsulation policy is applied to an object. Note that our implementation ensures that ObjectReferences are never nested. We are also experimenting with a cache that stores ObjectReferences so that they can be reused if the same encapsulation policy is applied to a particular object more than once.

[6]The reference values measured using the original Squeak virtual machine are 87 252 897 byte-codes/sec and 2 465 693 sends/sec.

The overall increase in memory due to OOE depends on how encapsulation policies are used. Since we have not yet finished a consistent refactoring of the Squeak image using OOE, we are not yet in a position to quantify it.

## 7. EVALUATION

In Section 2.4, we indicated that our goal was to define an encapsulation mechanism that was expressive, simple, and appropriate for dynamic languages. In this section we examine how well OOE meets these goals.

By way of assessment, we will look again at the problems from Section 2. If the problem can be solved at all, we can conclude that OOE is adequately expressive. Determining if the solution is simple and appropriate is more subjective, and here is it possible that the reader may reach different conclusions from the authors.

### 7.1 The Problem of Interdependence

Class interdependence caused by inheritance and instantiation can be controlled using OOE to the exact degree required. Returning to the example of the check method (see Section 2.2.1), recall that we wished to introduce an internal method check into a class C for the purposes of refactoring.

OOE allows us to make sure that this internal method is not accidentally overridden and called in any clients, whether they are subclasses or users of instances of C. This is done by making sure that the new method check is not included in any of the existing encapsulation policies that are offered by the class C. Note that although this completely hides check from all existing clients, OOE still allows us to make this method available to future clients by offering a *new* encapsulation policy, for example fullExtend, that grants access to the method check. Furthermore, if the override right **o** is explicitly *removed* from the policy that C uses to inherit from its superclass B, then C does not accidentally override a method with the same name even if it is introduced into B (and declared as overridable) at a later date.

Is this solution simple, and is it appropriate for a dynamic language? On the surface, it certainly does not seem simple when compared to an *ad hoc* solution such as labeling check as **private**. However, such an annotation does not seem to fit into the semantics of a dynamic language. Given the variety of different scenarios in which a class can be used, it indeed seems that to give it a systematic semantics, one would need to invent a way of defining multiple interfaces for a single class — in other words, we would need to invent a model containing something very like encapsulation policies!

Thus, we argue that we have in fact achieved *conceptual* simplicity. OOE uses a conceptually simple and uniform way to

specify access rights, and it gives them a simple and statically observable semantics that makes it easy to understand and reason about the code. Following the Smalltalk practice of integrating rich tool support into the programming environment, it is now a matter of designing appropriate *tools* to improve the practicality of our approach. For instance, we could provide usability similar to that of keyword-based techniques if an improved browser displayed each method together with the corresponding encapsulation attributes and allowed the programmer to change these attributes directly in this display.

OOE specifically addresses the needs of incremental development where requirements are changing and cannot always be narrowed down in advance. This is because it allows the class that provides some behaviour to offer its future clients a *range* of encapsulation policies from which to choose. Thus, an appropriate policy can be chosen at the time that the behavior is reused, not at the time that it is defined. In other words, encapsulation decisions are *late-bound*, which we feel is entirely appropriate for a dynamic language.

## 7.2 The Problem of Fragile Data Structures

The problem of fragile data structures can also be solved using encapsulation policies, as we have already shown in Section 4.5.2 for the submorphs example. Is the solution simple and appropriate? We have added a single language element to Smalltalk, selfWithPolicy:, which returns a reference to self with a different set of access rights. This is the *only* way in which access rights can be amplified, and because it gives rights only to self, it can be fully encapsulated. Our approach also seems appropriate for the dynamic and lightweight character of the language because it does not require type annotations; instead, changing the interface of an object reference is performed by sending a message like asReadonly.

It is true that this approach places on the programmer the burden of defining the necessary interface amplification or restriction methods — methods like asReadOnly and asEnumeration. More seriously, the programmer must also ensure that the rights granted by the various methods and the methods included in the various policies correspond. For example, if the **enumeration** policy were to accidentally contain the right to call asReadOnly, the distinction between **readOnly** and **enumeration** would be effectively lost, since any user possessing the enumeration right could also acquire the **readOnly** right.

Without ignoring the dangers of such accidents, we believe that they indicate a need for *tool support* rather than a conceptual shortcoming. It is easy to imagine a tool that would detect when a restrictive policy contained a method that gave access to a more liberal policy. More generally, a tool that lets the programmer define a lattice of encapsulation policies, either by writing constraints or by drawing a diagram, could construct the appropriate policy definitions and conversion operations and ensure that they correspond. We see a similarity here with the various tools that automatically generate instance variable access methods. Because in both cases the underlying semantics is simple (rights associated with references in the one case, and the fully protected nature of instance variables in the other), the programmer soon becomes accustomed to defining the appropriate methods, and does not feel the need to extend the language with features such as public instance variables.

# 8. RELATED WORK

The concept of *Composable Encapsulation Policies* as a flexible way of specifying and managing the access rights of methods was introduced by Schärli *et al.* at ECOOP 2004 using a set-theoretic and programming language-independent model [27]. The ECOOP paper focuses on the limitations of keyword based approaches and shows how they are overcome by encapsulation policies. Furthermore, although it contains a proposal for specifying and managing encapsulation policies in Smalltalk, it does not discuss how using such encapsulation policies could or should affect the *meaning* of a Smalltalk program. Specifically, the prior work does not address the meaning of the different access rights in a dynamically typed language, nor does it suggest that encapsulation policies could be used to solve common object encapsulation problems by associating encapsulation policies with object *references* rather than with objects.

MUST [30] is another encapsulation model that has been proposed for the language Smalltalk. The main difference from our model is that it is only concerned with module encapsulation and that it is much more static. Unlike OOE, MUST does not allow encapsulation decisions to be "late bound", that is, postponed until an individual client is about to use the encapsulated structure. Instead, it requires all clients in a predefined category (*i.e.*, superclasses, subclasses, unrelated classes) to access the class through the same interface. Furthermore, it requires the *implementor* to decide between different forms of self-sends (*e.g.*, locally-bound self-send and regular self-send) when a class is written. In contrast, OOE has only one form of self-send and allows each *client* of a class to decide how they should be bound by selecting an appropriate encapsulation policy.

The developers of Self also proposed an encapsulation model that addressed module encapsulation problems [12]. Although Self is a prototype-based language, the Self proposal is similar to the approach based on the dynamic type of the receiver discussed in Section 5.1.2. As a consequence, it is hard to implement efficiently. Another major difference from OOE is that the Self encapsulation proposal deals only with the question of which method can be called from where, and not the harder problem of choosing between re-implementing and overriding.

The Jigsaw modularity framework, developed by Bracha in his doctoral dissertation [10], defines a variety of module composition operators that control how attributes of a module are encapsulated. These operators give a programmer fine-grained control over *module* encapsulation, and allow the client to customize the access rights. However, the semantics of static binding and re-implementation is not based on the distinction between self-sends and object-sends. Furthermore, there is no support for addressing *object*-encapsulation problems.

Hogg *et al.* note that object aliasing has been recognized as a problem in both practical programming and formal verification for many years [18]. They also explain why many

existing solutions to the aliasing problem tend to be too conservative to be useful in practice. Subsequently, there has been a lot of work on new and more flexible solutions to aliasing problems. Some of these solutions use various forms of type annotations on pointer variables. An early form of this concept is the C++ keyword **const**, which prevents a reference from being changed. Other, more sophisticated, models provide annotations to declare a reference as *unique* (*i.e.*, it is the only reference to the object) and *borrowed* (*i.e.*, it is a reference that may not be stored into an object's instance variables) [3, 9, 17, 20]. Boyland gives an excellent overview and a comparison of the different annotation-based proposals [9], and suggests that annotations should not be considered individually but as a part of a general capability system for pointers.

Although these proposals are, like OOE, based on defining access rights to object on the granularity of references, there are many differences. One of the most significant differences is that they are based on annotating pointer variables with a *fixed* set of predefined access rights, such as read, write, and exclusiveRead. In contrast, OOE controls which *messages* can be passed to an object reference, allowing us to obtain object references with an *unlimited* set of customized access rights, which we represent as encapsulation policies.

While this difference makes reasoning about capabilities easier and allows them to model certain properties (*e.g.*, *uniqueness*) that cannot be modeled using encapsulation policies, they are fundamentally not object-oriented — unless one's universe of objects understand only two messages, read and write. OOE is integrated into the message-based character and semantics of dynamically typed languages like Smalltalk, and it avoids type annotations, which do not feature in dynamic languages. Furthermore, OOE addresses both object and module encapsulation problems in a uniform way, while capabilities address object encapsulation only.

Ownership types are another approach to the aliasing problem [8, 13, 25]. Unlike capabilities and OOE, the basic concept of this approach is to prevent aliases rather than controlling what can be done through them. On the one hand, this has the advantage that one can state that a certain object should be owned by an aggregate and is therefore automatically protected (by a sophisticated type system) from being passed outside. This avoids the problem that one might accidentally pass out an unprotected reference, and makes the approach easy to reason about. On the other hand, these same safety guarantees make these approaches more restrictive, and even recent suggestions for flexible alias protection still do not allow one to implement many commonly used data structures [20]. Barnett and Naumann tackle some of these limitations by using friendship systems that allow state dependence across ownership boundaries [5].

Noble *et al.* proposed an encapsulation model based on object ownership for Self [24]. In comparison to OOE, this model does not provide help for module encapsulation problems and is too restrictive to express some commonly-used programming patterns such as external iterators. Where it can be applied, it gives the programmer more guarantees, but one has to expect a relatively high runtime overhead, particularly for the argument rule.

The delegation-based programming language *E* [14] features *facets*, which restrict the methods that can be sent to a certain object. Although facets and encapsulation policies have a similar purpose, there are several differences. Facets are written as wrapper objects that simply delegate the valid method calls to the real object. This means that in contrast to encapsulation policies, facets contain actual code. Furthermore, facets only control whether a method can be called, but not whether it can be re-implemented or overridden. This is related to the fact that *E* is based on delegation and simulates traditional inheritance by parameterizing the object constructor with an argument to contain **self**.

# 9. SUMMARY AND FUTURE WORK

We have introduced an encapsulation model for dynamically typed languages that addresses both module encapsulation problems (*i.e.*, interdependencies between different classes) and object encapsulation problems (*i.e.*, fragile data-structures) in a uniform way. Our model is based on two cornerstones. First, we use encapsulation policies to capture all of the encapsulation aspects of both *classes* and *objects*, which makes our model uniform and conceptually simple. Second, we define the meaning of message passing in the presence of encapsulated objects by introducing the distinction between two conceptually different kinds of message send: "internal sends", which are used for sending a message to oneself without crossing the encapsulation boundary, and "external sends", which are used for sending a message to another object through the associated encapsulation policy.

In combination, these two cornerstones make our model particularly appropriate for dynamically typed languages. This is because of its conceptual simplicity and uniformity, and the absence of type annotations. Instead, it relies on sending messages to objects both for defining and for controlling the operations that can be accessed through individual object references.

We have presented our model in terms of the language Smalltalk and illustrated it with various examples. We believe that it is equally applicable to other dynamic object-oriented languages such as Ruby. We have given a detailed discussion of the motivation for our design decisions, in particular with regard to static understandability of the program code, and we have given a detailed description of our implementation in Squeak Smalltalk. We also presented different benchmarks that show that the performance penalty imposed by our model is moderate.

We are about to refactor more Squeak code using OOE. This will serve as the basis for a more detailed evaluation of the practicality of our approach. We also want to experiment with applying our model to traits [26]. This looks like a very promising combination because our encapsulation model has been designed to be well-suited to non-standard composition mechanisms, and in particular to multiple composition of behavior. Furthermore, as mentioned in Section 7, we plan to provide tool support for constructing and manipulation of encapsulating encapsulation policies, for expressing recurring encapsulation patterns more easily, and for automatic detection of possible "encapsulation holes".

## Acknowledgments

The work described in this paper has been partially supported by the Swiss National Foundation under the project "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004), and by the National Science Foundation of the United States under awards CCR-0098323 and CCR-0313401. We are grateful to our colleagues at SCG and OGI for valuable discussions, and in particular to Phil Quitslund for drawing the diagrams.

## 10. REFERENCES

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chan, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, 1995.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings OOPSLA 2002*, pages 311–330. ACM Press, Nov. 2002.

[3] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proceedings ECOOP '97*, pages 32–59. Springer Verlag, June 1997.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[5] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Proceedings MPC 2004*, July 2004. To appear.

[6] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[7] E. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *Proceedings ECOOP '87*, pages 41–50. Springer-Verlag, June 1987.

[8] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings POPL'03*, pages 213–223. ACM Press, 2003.

[9] J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In *Proceesings ECOOP 2001*, pages 2–27. Springer Verlag, June 2001.

[10] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, Mar. 1992.

[11] K. B. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[12] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *Lisp and Symbolic Computation*, 4(3):207–222, July 1991.

[13] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64. ACM Press, 1998.

[14] The E Language. http://www.erights.org/.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[16] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[17] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings OOPSLA '91*, volume 26, pages 271–285. ACM Press, Nov. 1991.

[18] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[19] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, Nov. 1997.

[20] G. Kniesel and D. Theisen. JAC - access right based encapsulation for Java. *Software - Practice and Experience*, 31(6):555–576, May 2001.

[21] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[22] O. Nierstrasz. A survey of object-oriented concepts. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison Wesley, Reading, Mass., 1989.

[23] J. Noble. Iterators and encapsulation. In *Proceedings of TOOLS '00*, page 431ff, June 2000.

[24] J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *Proceedings TOOLS '99*, Nov. 1999.

[25] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. In E. Jul, editor, *Proceedings ECOOP '98*, Brussels, Belgium, July 1998. Springer Verlag.

[26] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, pages 248–274. Springer Verlag, July 2003.

[27] N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In *Proceedings ECOOP 2004*. Springer Verlag, June 2004. To appear.

[28] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86*, pages 38–45. ACM Press, Nov. 1986.

[29] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings OOPSLA '96*, pages 268–285. ACM Press, 1996.

[30] M. Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *IEEE Software Engineering Journal*, 7(2):95–102, Mar. 1992.