Composable Encapsulation Policies*

Nathanael Schärli[1], Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts[2]

[1] Software Composition Group
University of Bern
www.iam.unibe.ch/∼scg
[2] Lab for Software Composition and Decomposition
Université Libre de Bruxelles
http://homepages.ulb.ac.be/∼rowuyts/

Abstract. Given the importance of encapsulation to object-oriented programming, it is surprising to note that mainstream object-oriented languages offer only limited and fixed ways of encapsulating methods. Typically one may only address two categories of clients, users and heirs, and one must bind visibility and access rights at an early stage. This can lead to inflexible and fragile code as well as clumsy workarounds. We propose a simple and general solution to this problem in which encapsulation policies can be specified separately from implementations. As such they become composable entities that can be reused by different classes. We present a detailed analysis of the problem with encapsulation and visibility mechanisms in mainstream OO languages, we introduce our approach in terms of a simple model, and we evaluate how our approach compares with existing approaches. We also assess the impact of incorporating encapsulation policies into Smalltalk.

1 Introduction

Encapsulation is widely acknowledged as being one of the cornerstones of object-oriented programming [Nie89]. Nevertheless, the term *encapsulation* is often used in inconsistent ways.

At the very least, encapsulation refers to the *bundling together of data and the operations that manipulate them*. That is, *information hiding* is not necessarily an essential component of encapsulation. At the same time, the terms *encapsulation boundary* and *violation of encapsulation* suggest that information hiding is typically, albeit not necessarily, implied by encapsulation. In practice, depending on the programming language in use, or the programming conventions being applied, different *policies* concerning encapsulation may be in effect.

Snyder, in a classic paper [Sny86], defines encapsulation as follows.

> 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.

* In Proceedings ECOOP 2004, LNCS 3086, pp. 248–274, Springer Verlag, 2004

We feel this definition captures the essence of encapsulation, but we observe (as did Snyder) that present day object-oriented programming languages are surprisingly weak in terms of the mechanisms they offer programmers to establish the encapsulation policies that a class offers to its clients. In particular, we identify the following three weaknesses as being endemic to OO languages:

1. *Access rights are inseparable from classes:* access rights to methods are specified as part of their implementation. As a consequence, it is neither possible to apply the same policies in a reusable way to different classes, nor is it possible to apply different policies to the same class.
2. *Client categories are fixed:* existing languages offer only the possibility to specify access rights for a fixed set of client categories, typically users and heirs (*i.e.*, instances using a fixed, public interface, and subclasses).
3. *Access rights are not customizable:* the onus is on the provider to specify the access rights. It can be hard or impossible for a client to adapt certain encapsulation decisions once they are fixed by the providing class.

We propose to address these problems by turning encapsulation policies into separate entities that can be composed. We characterize our approach as follows:

- A class consists of an implementation and a number of *encapsulation policies*.
- An encapsulation policy is a mapping from method signatures to *access rights*.
- The set of access rights may be language-specific, but will typically express whether a method may be *called*, *implemented* or *overridden*.
- Encapsulation policies can be *composed*. One may *merge* available policies, thus combining their access rights, or *refine* a policy to obtain a more restrictive one.
- A client can use a class through a default encapsulation policy, explicitly select one of the available policies, or a specify a customized policy.

We claim that this simple model of composable encapsulation policies addresses the weaknesses we have identified above in a fundamental way. Encapsulation policies not only give the programmer freedom to specify multiple usage contracts for different classes of clients, but they allow certain critical decisions to be delayed until the client is ready to bind them. Furthermore, encapsulation policies subsume other, less general, mechanisms, such as interfaces, and visibility mechanisms.

The contributions of this paper include:

- an analysis of the weaknesses in the encapsulation mechanisms of mainstream OO languages,
- a proposal for a new encapsulation mechanism based on composable encapsulation policies,
- a simple formalization of this mechanism,
- a detailed discussion of how we applied this model to Smalltalk,
- an evaluation of the proposed mechanism, including a comparison with mainstream OO languages.

This paper is structured as follows: In section 2 we motivate this work by presenting a detailed analysis of the shortcomings of present encapsulation mechanisms. In section 3 we propose encapsulation policies by means of a simple, set-theoretic model. In section 4 we present how this model can be applied to Smalltalk based on the experiences with our prototype implementation. We then evaluate our proposal with respect to the identified problems and provide some discussion in section 5. We review related work in section 6, and we conclude in section 7 with some remarks on future and ongoing work.

2 Problem Statement

In this section, we motivate our work by analyzing the limitations of the encapsulation mechanisms offered by mainstream object-oriented programming languages such as Java, C++, C#, and Eiffel.

2.1 Access Rights are Inseparable from Classes

In most object-oriented languages, the access rights to classes are tightly bound to their implementation. In languages like Java, C++, and C#, methods may be annotated with certain access rights by using keywords such as public, private or protected. Since the access rights are inseparable from the methods they are applied to, they cannot be reused independently. It is consequently impossible to express, for example, that methods >, <, <=, *etc.* should be public in all classes that implement the magnitude protocol. Instead, the programmer has to express this information on a per-method basis and duplicate it in each class that implements these methods.

Illustration. As a concrete example, consider the two classes Collection and Path that each implement a collection protocol which typically consists of a few dozen methods such as add:, addAll:, remove:, do:, and select:. Path inherits from GraphicalObject and Collection inherits from Object, so they are not related by inheritance. In current languages, *both* of these classes must individually specify their encapsulation attributes for these methods (*i.e.*, which should be public, private, or protected). It is not possible to express these attributes in a sharable way.

The situation is worse if there are subclasses of Collection and Path in which only a subset (or a superset) of these methods should be accessible to a client, because the programmer is again forced to specify the new access rights on a per-method basis and duplicate this information to make it available in both subclasses.

2.2 Client Categories are Fixed

Most object-oriented languages such as Java and C# offer a set of keywords (*i.e.*, private, public, and protected) that essentially allow the designer of a class to assign encapsulation policies for just two fixed categories of clients (*i.e.*, users

and heirs) corresponding to two different modes of use (*i.e.*, instantiation and inheritance) [Sny86].

This approach restricts modularity because it does not take into account that different clients within the same category may need to access a class in different ways [Bra92]. By forcing the designer of a component to fix the encapsulation policy for each category of client, one takes away the freedom of the client to choose which mode of use is more appropriate, and one loses the ability to distinguish between different needs of clients within such a category.

Illustration. The class Morph is the root of all the graphical objects in the user interface framework of Squeak [IKM+97] and implements several hundred methods, most of which are internal auxiliary methods. Since this framework is designed to be extended by inheritance, Morph has many subclasses with a variety of different needs for encapsulation.

The vast majority of subclasses, exemplified by SketchMorph, specialize how the Morph is drawn on a graphical canvas. This means that they typically override only a very small set of designated hook methods such as drawOn: and drawPostscriptOn:, which are then called by other methods such as fullDrawOn: and refreshOn: that are part of the "drawing protocol" of Morph. However, there are also other kinds of subclasses that override more than these hook methods. The class PluggableListMorph, for example, specializes other drawing methods to implement smooth scrolling and overrides the submorph management methods since it uses a list as a model and therefore does not need to explicitly store submorphs.

Several other kinds of subclasses of Morph exist, and each needs to customize certain methods of Morph. But unfortunately, encapsulation models like the one of Java are not expressive enough to address the needs of these different categories of subclasses. Instead, the designer of the class Morph has to declare practically all internal methods as public or as protected, in order not to restrict the most demanding clients from specializing the functionality of Morph according to their needs.

However, such a "one-size-fits-all" encapsulation policy is not appropriate for the majority of subclasses that just want to customize some designated hook methods or add some special purpose methods. This is because it *forces* all these subclasses to access the class through an extremely wide and error prone interface that unnecessarily restricts their freedom of choosing names for local auxiliary methods and makes them unnecessarily fragile with respect to changes in Morph (*e.g.*, introducing a new protected method in Morph will break any subclass that incidentally uses the same name for an own internal method).

This fragility is *unnecessary* because most subclasses neither need nor want to override any of these protected methods in Morph, but because the same encapsulation policy must be shared by all subclasses, they cannot explicitly declare this.

Existing Solutions. Eiffel addresses this problem by allowing the designer of a class to declare the classes that are allowed to access a certain method. However,

this solution is very limited, because the designer has to take an up-front decision on the clients that will have access. Clients that are not known when the class is written (and are not subclasses of known clients) cannot be taken into account and so can never have access. Furthermore, it cannot be used to discriminate between different heirs, which means that all heirs access the class through a completely unrestricted interface.

C++ addresses this problems with the friend construct that allows a class to grant other functions or classes access to its internal members. Like the Eiffel approach, this is a very limited solution because the clients have to be known upfront. Furthermore, it is not fine-grained enough because a friend is always allowed to access *all* the otherwise private methods and fields, without distinction. Similarly, private and protected inheritance do not generally solve these problems: they allow a programmer to make either all or none of the methods available in a subclass, but are not fine-grained enough to address the precise needs of different subclasses.

In Java and C#, methods and fields can be defined to be accessible within the current package and the current assembly, respectively. However, this approach is also not flexible enough because it allows programmers to establish only one additional category of clients, which is given by the physical organization of classes and underlies therefore many constrictions. For example, each Java class can only be part of exactly one package.

2.3 Access Rights are not Customizable

It is clear that it is primarily the responsibility of the designer to define how a class should be encapsulated. But even if a language allowed the designer to specify an arbitrary number of encapsulation policies, it would neither be possible nor reasonable for the designer to provide a policy that precisely addresses the individual needs of each client. Therefore, a language should allow a client to customize the encapsulation policy according to its individual needs as long as it does not violate the restrictions defined by the designer of the component. This means that a client should be allowed to make the interface granted by an encapsulation policy smaller, but not larger.

Unfortunately, current languages offer at best only limited support for such customization. Java, for example, does not allow the client of a class to customize the encapsulation policy specified by its designer, whereas C++ offers only very coarse-grained and limited mechanisms (*i.e.*, public, private, and protected inheritance). This not only prevents unanticipated reuse, it also prevents a client from using a class through a customized encapsulation policy that minimizes the risk of inappropriate method accesses and reduces fragility with respect to changes in the used class.

To avoid unnecessarily fragile class hierarchies, the programmer of a subclass should, for example, have the means to decline the override right for all methods but the ones that effectively need to be overridden. This makes the new subclass invulnerable to the problem of unintended name clashes that can occur when the implementor of a superclass changes its internal implementation and adds

some new auxiliary methods. Even if one of the new methods incidentally has a signature that is already used in the subclass, the absence of the override access right guarantees that the methods do not interfere.

Illustration. Consider a C++ class Point where the method == is implemented by comparing the two coordinates x and y. Furthermore consider a method moveTo(Point) that uses == and is implemented as follows:

```
void moveTo(Point other) {
    if (this == other) return;
    x = other.x;
    y = other.y;
    coordinatesChanged();   // Notify my clients
}
```

To allow another programmer to override the method == in a specialized subclass such as LargeIntegerPoint, == is declared as virtual, which grants subclasses the right to override this method. But since this encapsulation policy cannot be adapted by the subclass, it does not only *allow* a subclass to override this method, but it also *prevents* subclasses from accessing the method in another way; once the designer of Point has decided that this method will be dynamically bound, heirs can no longer customize this decision and make it statically bound.

In particular, this means that it is not possible for a client to implement a new method == without having all the calls to == in Point be bound to this new method. As a consequence, this encapsulation decision significantly restricts the freedom of all direct and indirect subclasses of Point because it does not allow them to use the method == in a way that does not fully conform to the original implementation.

It is for example not possible to implement a subclass ColoredPoint of Point where == takes into account both the color and the coordinates without breaking moveTo and all the other methods in Point that call == and expect that it just compares the coordinates.

Existing Solutions. Both Eiffel and C# address this problem and allow a subclass to resolve such unintended name captures. In Eiffel this is done by allowing a subclass to consistently rename arbitrary methods of the superclass. C# allows the programmer to assign the keyword new (rather than override) to a method to declare that it is used for a different concept than in the superclass and that all calls in the superclass should therefore be statically bound to the local method.

However, these solutions are not as flexible as they should be. Eiffel only allows the subclass to *resolve* unintended name captures that are apparent when the subclass is written, but it does not allow the subclass to *protect* itself from unintended name clashes that may occur later, for instance when the superclass is modified and new methods are added. This is because only existing superclass methods can be renamed in a subclass.

In C#, this limitation is avoided because the keywords new and override can also be used for methods that do not (yet) have a corresponding method in

the superclass[3]. However, the C# approach suffers from the same limitations as described in section 2.1, which means that the only way to protect internal methods from such unintended name clashes is to explicitly assign the keyword new to the *implementation* of each of these methods. It is not possible for a programmer to declare a *reusable* policy that declines the override right for *all but* the methods where this right is effectively needed and then share this policy among a family of subclasses that need to override the same superclass methods but may use different internal methods that should all be protected from unintended name clashes that can arise when the superclass is modified.

Another limitation is that these solutions only allow a subclass to decline the right to override a method, but they do not allow *any* client to decline *any* access right that is granted by an encapsulation policy. Thus, it is for instance not possible for a subclass to declare that certain internal superclass methods cannot be called because they are inappropriate in a specific usage scenario.

3 Encapsulation Policies

In this section we present a new model for specifying encapsulation policies for object-oriented programming languages. We use a simple, set-theoretic approach to describe the model in a language-independent way. For concreteness, we use the terminology of class-based languages with inheritance and instantiation as the only two modes of use for a class. Note however that the concept of composable encapsulation policies is very general and could also be applied to prototype-based languages as well as languages that support non-standard composition mechanisms such as automated delegation [VRB00] or trait composition [SDNB03].

3.1 Design Rationale and Overview

As we have seen in section 2, most of the weaknesses in present encapsulation mechanisms arise from the fact that encapsulation policies are inseparable from the implementation. We propose to tackle this problem essentially by introducing encapsulation policies as separate entities, which can be individually selected, composed and applied. We apply the following principles:

- An *encapsulation policy* expresses how a client can access the methods of a class, independent of the particular mode of use (*i.e.*, instantiation or subclassing).
- The designer can associate an arbitrary number of encapsulation policies to a class. Each policy represents a set of encapsulation decisions that correspond to a certain usage scenario.
- The client can independently decide which encapsulation policy to apply and in which way the class will be used. The chosen policy may be one that is provided by the class, or one that is *stricter* than a provided one.

[3] This causes a compiler warning but not an error.

Note that we only consider methods in the encapsulation policy, since we assume that instance variables are never accessible from the outside of an object.

3.2 Modelling Encapsulation Policies

We now present a simple model of encapsulation policies.

An *encapsulation policy* $P : \mathcal{S} \mapsto 2^{\mathcal{A}}$ is a mapping from method signatures to potentially empty sets of access attributes. P represents a contract between the class and its client. This means that a client accessing a class through P may only access a method m with signature s according to the set of attributes $P(s)$.

A signature $s \in \mathcal{S}$ identifies a method provided by the class. This may simply represent a method name, for dynamically typed languages like Smalltalk, or might include type information for statically typed languages with overloading, like Java and C++.

The access attributes represent the policy in effect that constrains how clients may use the method. The actual set of available access attributes may depend on the particular programming language. For the purpose of illustration, we will consider three kinds of access attributes, namely **c**, **r** and **o**, which specify, respectively, that the associated method may be *called*, *reimplemented* or *overridden*.

A word of explanation may be in order. We draw an important distinction between *reimplementing* and *overriding* a method in a subclass. If a subclass overrides a method, this means that all existing calls to this method in the superclass are *dynamically bound* to the overriding method. If a subclass does not override but only reimplements a method, existing calls in the superclass continue to be *statically bound* to the old version of the method. In Java, for example, a subclass may reimplement a method that has been declared as private in its superclass, but one cannot override it — the new method is not visible from the context of the superclass and all the calls remain statically bound to the local version of the method. By contrast, a subclass can neither reimplement nor override a method that is declared as final in its superclass.

Access attributes express rights that are conceptually *orthogonal*. We consider the access rights **c**, **r** and **o** to be orthogonal since each can logically occur in isolation independently of the others, whether or not all combinations are sensible or desirable. In most programming languages, only certain combinations may make sense, or might be expressible using the mechanisms available. For example, protected in Java corresponds to the rights $\{\mathbf{c}, \mathbf{o}\}$ — a heir may call protected methods and may override them, but may not simply reimplement them.

3.3 Composing Encapsulation Policies

We now define operators and relations over encapsulation policies that enable us to compose them and express constraints on their composition.

Suppose that P and Q are arbitrary encapsulation policies and s is an arbitrary method signature. Then we define the following:

– The policy $P + Q$ is the *merge* of P and Q:

$$(P + Q)(s) \stackrel{\text{def}}{=} P(s) \cup Q(s)$$

– The policy $P * Q$ is the *intersection* of P and Q:

$$(P * Q)(s) \stackrel{\text{def}}{=} P(s) \cap Q(s)$$

– The policy $P - Q$ is the *reduction* of P by Q:

$$(P - Q)(s) \stackrel{\text{def}}{=} P(s) - Q(s)$$

– For a set of selectors $S \subseteq \mathcal{S}$, the policy $P|S$ is the *restriction* of P to S:

$$(P|S)(s) \stackrel{\text{def}}{=} \begin{cases} P(s) \text{ if } s \in S \\ \emptyset \quad \text{otherwise} \end{cases}$$

– P is *stricter* than Q if all rights granted by P are also granted by Q:

$$P \leq Q \stackrel{\text{def}}{\Leftrightarrow} P(s) \subseteq Q(s), \forall s \in \mathcal{S}$$

– $P[a]$ is the set of method signatures for which right $a \in \mathcal{A}$ is granted:

$$P[a] \stackrel{\text{def}}{=} \{s \in \mathcal{S} \mid a \in P(s)\}$$

– The policy $P \setminus A$ is the result of *removing* the access rights $A \subseteq \mathcal{A}$ from P:

$$(P \setminus A)(s) \stackrel{\text{def}}{=} P(s) - A$$

3.4 Encapsulation Constraints

In class-based languages, clients use classes via two kinds of operations: inheritance and instantiation. With our approach, both of these operations are parameterized with an encapsulation policy that imposes certain constraints on the client.

Inheritance. Consider a chain of subclasses $C_0, C_1, \ldots, C_n$ where C_0 is the root of the class hierarchy, C_n is a concrete class, and the class C_i is defined as the subclass of C_{i-1} using the encapsulation policy P_i, for all $i \in \{1, \ldots, n\}$. For any class C, the term $pol(C)$ denotes the set of encapsulation policies offered by C, and $meth(C)$ denotes the set of methods implemented in C. Furthermore, we use $sig(C)$ to denote the signatures of the methods in $meth(C)$, we use $self(C)$ to denote the set of signatures that are sent to `self` in any of the methods in $meth(C)$, and we use $super(C)$ to denote the set of signatures that are sent to `super` in any of the methods in $meth(C)$.

For the concrete class C_n to be *valid*, the following encapsulation constraints must be fulfilled for all $k \in \{1, \ldots, n\}$.

$$\exists Q \in pol(C_{k-1}) : P_k \leq Q \quad (1)$$
$$sig(C_k) \cap \bigcup_{i<k} sig(C_i) \subseteq P_k[\mathbf{r}] \cup P_k[\mathbf{o}] \quad (2)$$
$$self(C_k) \cap \bigcup_{i<k} sig(C_i) \subseteq \bigcup_{n \geq i \geq k} sig(C_i) \cup P[\mathbf{c}] \quad (3)$$
$$super(C_k) \subseteq P[\mathbf{c}] \quad (4)$$
$$\forall Q \in pol(C_k) : Q|(\mathcal{S} - sig(C_k)) \leq \sum_{Q' \in pol(C_{k-1})} Q' \quad (5)$$

The first constraint guarantees that the policy P_k through which the client C_k uses the class C_{k-1} can only grant access rights that are also granted by a certain encapsulation policy Q offered by C_{k-1}. The second constraint makes sure that the class C_k only implements methods with signatures that are not defined in any of its superclasses, are allowed to be reimplemented by P_k, or are allowed to be overridden by P_k. The third constraint ensures that there are only self-sends to methods that are inherited from P_{k-1} if they are declared as callable by the policy P. Note that self-sends to methods implemented in C_k or one of its subclasses are always allowed, even if they have the same signature as a method implemented in a superclass of C_k. The fourth constraint ensures that there are only super-sends to methods that are declared as callable by the policy P_k.

Finally, the fifth constraint guarantees that for all the method signatures that are not implemented in C_k, the encapsulation policies $pol(C_k)$ offered by C_k can only grant access rights that are also granted by at least one of the policies $pol(C_{k-1})$ of the superclass C_{k-1}. This is important because it guarantees that the encapsulation restrictions that the designer of the class C_{k-1} assigned to its methods cannot be bypassed in indirect subclasses. Note that the subclass C_k is free to grant arbitrary access rights for all the methods $meth(C_k)$ that are implemented locally.

Note that these constraints do not prevent a subclass C_k from offering its clients a policy that grants more rights on the methods obtained from C_{k-1} than the policy P_k, through which the class C_k inherits from C_{k-1}. This is important because it allows a class C_k to access a class C_{k-1} through a *minimal* policy P_k without preventing its future clients from accessing the methods obtained from C_{k-1} through a policy that grants more rights. Also note that these constraints do not guarantee that the class C_k is correct (*i.e.*, that there are no calls to methods that are not available) nor are they concerned with issues related to subtyping (see section 5.4). Instead, they only ensure that the encapsulation restrictions are not violated.

Instantiation. Instantiation is also parameterized with an encapsulation policy, which means that each new instance o of the concrete class C_n is created through an encapsulation policy P. For such an instantiation and subsequent calls on o to the selector s to be *valid*, the following encapsulation constraints must be fulfilled.

$$\exists Q \in pol(C_n) : P \leq Q \quad (1)$$
$$s \in P[\mathbf{c}] \quad (2)$$

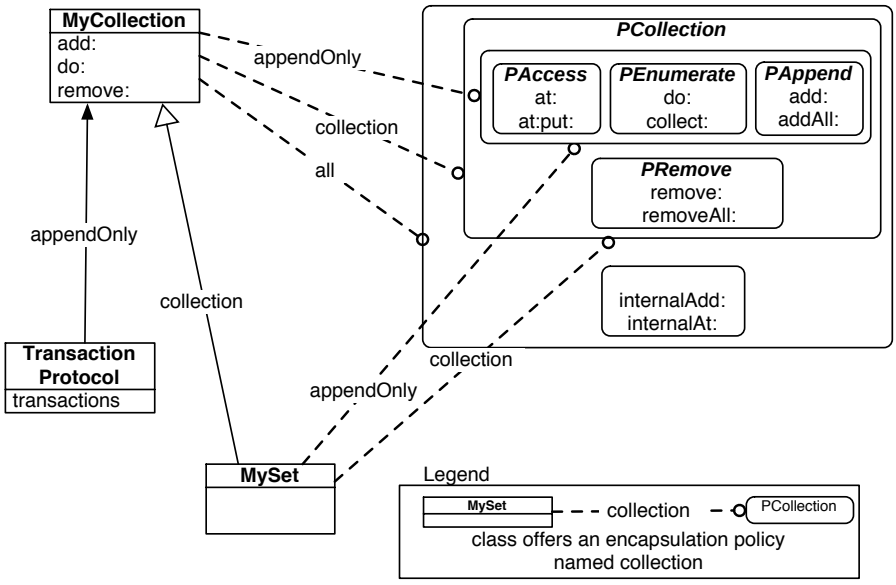

Fig. 1. Encapsulation policies at work

The first constraint is the same as for inheritance and it says that the policy P can only grant access rights that are also granted by a certain policy Q offered by C_n. The second constraint says that the only method signatures that are allowed to be called on the instance o are the ones that are marked as callable in P.

3.5 Example

In the example shown in figure 1, the class MyCollection offers three different encapsulation policies that are available under the names appendOnly, collection and all. Note that each of these three policies is composed from several stricter policies, some of which are shared.

The class MyCollection has two clients. One client is the subclass MySet, which inherits from MyCollection through the encapsulation policy collection and in turn offers the same policy as well as the policy appendOnly to its clients. The other client is the class TransactionProtocol, which uses an instance of MyCollection through the encapsulation policy appendOnly to store its transactions. To be usable, the class TransactionProtocol must also offer at least one encapsulation policy, but this is not shown in the figure.

4 Encapsulation Policies in Smalltalk

In the previous section we have introduced the model of encapsulation policies in a language independent way. Now we show how this model can be applied to Smalltalk. This section is based on our proof of concept implementation in the Smalltalk dialect Squeak [IKM+97]. However, we will present the examples in a somewhat simplified syntax to ease the reading of the paper, especially for readers who are not familiar with Smalltalk. In particular, we use bold face for symbols rather than the prefix `#`.

4.1 Representing Encapsulation Policies

Following the Smalltalk tradition of making everything an object, encapsulation policies are instances of the class Policy. Each policy object can contain some local definitions, which are represented as a dictionary of associations from method selectors (*i.e.*, symbols) to access attributes, and can refer to other encapsulation policies which it is composed from.

Creating Encapsulation Policies. In the previous section, we have pointed out that the actual set of access attributes may depend on the programming language. Since Smalltalk is dynamically typed and inheritance is often used for sharing implementation in unanticipated ways, we believe that it does not make much sense to declare a method that cannot be reimplemented in a subclass. Therefore, we define that a method can *always* be reimplemented, and consequently, our Smalltalk encapsulation policies only manage the access attributes callable (**c**) and overridable (**o**).

For convenience, we provide a literal way of creating encapsulation policies. This is done by putting the selectors between brackets [] and prefixing them with either ↑ or ↓ to indicate the associated access rights. The meaning of such a literal policy is defined as follows:

- No prefix means that the selector is fully accessible ({**c**, **o**})
- The prefix ↑ means that the selector is callable but not overridable ({**c**})
- The prefix ↓ means that the selector overridable but not callable ({**o**})
- All selectors that do not appear in the policy definition are neither callable nor overridable ({})

As an example, the expression [foo ↑bar ↓check] returns a policy that allows full access to the selector **foo**, allows the selector **bar** to be called but not overridden, and allows the selector **check** to be overridden but not called. All the other selectors are neither allowed to be called nor overridden. All selectors are allowed to be reimplemented.

Manipulating Encapsulation Policies. Since encapsulation policies are first-class objects, they can be manipulated by messages. If p and q are arbitrary encapsulation policies, the most common messages and their semantics are as follows:

- + is the merge operator, which means that the expression p + q returns a new policy that grants all the access rights granted by either p or q.

- * is the intersection operator, which means that the expression p * q returns a new policy that grants only the access rights that are granted by both P and Q.
- - is the reduction operator, which means that the expression p - q returns a new policy that grants the access rights granted by p without the rights granted by q.
- The expression p noOverride returns a new policy that is the same as p except that no selector can be overridden:

$$\mathsf{p\ noOverride} \equiv \mathsf{p} - \{\mathbf{o}\}$$

4.2 Associating Encapsulation Policies with Classes

A key feature of the model is that a programmer can associate an arbitrary number of encapsulation policies with a class. In our Smalltalk implementation, this is done by sending the message policyAt:put: to a class. This message takes a symbol and a policy as an argument and then associates the policy with the class under the identifier represented by the symbol. All the identifiers associated with encapsulation policies are local to the class, and they allow a client to refer to a certain encapsulation policy that is offered by the class. As an example, we can define a collection class OrderedCollection as follows:

```
(Collection subclass: OrderedCollection)
    instanceVariableNames: 'array offset';
    policyAt: basicUse put: [add: addAll: removeAt: ...];
    policyAt: basicExtend put: basicUse + [growBy: compact ...]
```

This creates the class OrderedCollection as a subclass of Collection, defines the two instance variables array and offset, and associates two encapsulation policies with the identifiers **basicUse** and **basicExtend**. Note that it is possible to define a policy that refers to another policy associated with the class by using its identifier in the policy definition. In our example, the expression **basicUse** + [growBy: compact ...] refers to the policy **basicUse** and merges it with the policy [growBy: compact ...].

The order of the policyAt:put: messages is only relevant as far as later messages override policies that have been bound to the same identifier by earlier messages. However, the order is not relevant for the meaning of the policy definition (*i.e.*, the second argument). This is because references to other policies are not evaluated when the expression is executed. Instead, these references remain part of the policy definition, which means that the relationship between the different policies remains valid even if one of the involved policies gets modified. Note that circular references in policy definitions are not allowed and result in an error.

4.3 Using Encapsulation Policies

When creating a subclass or an instance of class, the programmer can select which of the encapsulation policies offered by the class should be applied. This is done by passing the symbol selecting the encapsulation policy as an additional argument to the message that creates the new subclass or instance. The following code illustrates how the message newWithPolicy: is used to create a new instance of the class OrderedCollection using the encapsulation policy **basicUse**:

```
Morph>>initialize
    super initialize.
    submorphs := OrderedCollection newWithPolicy: basicUse.
    ...
```

As a consequence, the ordered collection submorphs responds only to the messages that are declared as callable by the policy **basicUse** in the class OrderedCollection. Sending any other messages leads to a runtime error.

Default Policies. To improve ease of use without sacrificing the flexibility of having multiple encapsulation policies, our implementation features the concept of *default policies*. The designer of a class can specify two default policies for a class by associating ordinary encapsulation policies with the identifiers **basicUse** and **basicExtend**. When a client uses this class and does not explicitly specify another policy, these default policies are then used for creating new instances and subclasses, respectively. This means that in the previous example, we could have used the simpler expression OrderedCollection new to create an instance that implicitly uses the default policy **basicUse**.

4.4 Sharing Encapsulation Policies

Although it is possible to define anonymous policies using the [] notation, it is often more appropriate to declare *named policies* and then share them among different classes. Because of the lack of namespace facilities in Squeak, we store policies in the same namespace as classes and just use the convention that we prefix policy names with the letter P.

In the following example, we first define encapsulation policies named PEnumeration, PAppend, and PRemove. Then we merge these policies to define a policy named PCollection, which is then shared between the classes Collection and Path described in section 2.1.

```
Policy named: PEnumeration
    is: [do: select: detect: collect: reject: ...].
Policy named: PAppend
    is: [add: addAll: ...].
Policy named: PRemove
    is: [remove: removeAll: ...].
Policy named: PCollection
    is: PEnumeration + PAppend + PRemove.
```

```
(Object subclass: Collection)
    instanceVariableNames: '';
    policyAt: basicUse put: PCollection

(GraphicalObject subclass: Path)
    instanceVariableNames: 'points';
    policyAt: basicUse put: PCollection + [draw drawOn: length segmentCount ...]
```

Note that neither classes Path nor Collection specify a policy to access their respective superclass, which means that the default policy **basicExtend** is applied.

Special Policies. The policy PProtoObject is defined so that it allows all the methods that are implemented in the class ProtoObject to be called. To guarantee that every object responds at least to this common set of system messages such as == and isNil, this policy is implicitly added to any policy that is assigned to a class (*e.g.*, using the message policyAt:put:). This means that the policy of the class Collection in the above example is in fact equivalent to PCollection + PProtoObject.

Traditionally, all methods in Smalltalk are public, and therefore many Smalltalk programmers enjoy the freedom of not having to deal with encapsulation decisions if they do not want to. We support this style of programming with a policy PAll, which allows full access to any valid method selector. As a consequence, it is possible to make a class fully accessible from the outside by simply associating the policy PAll to the default identifiers **basicUse** and **basicExtend**.

4.5 Encapsulation Policies in Subclasses

In section 3.4, we have formally defined the constraint that applies to encapsulation policies offered by subclasses (constraint 5). In our implementation, we ensure this by implicitly restricting each policy that is assigned to a class (*e.g.*, using the message policyAt:put:) so that it does not grant any access right for inherited methods that are not also granted by the union of the policies offered by the superclass.

To allow a programmer to create subclasses that have less encapsulation policies than their superclass, policies that are assigned to a class are not automatically available in subclasses. However, a programmer can "inherit" the policies offered by a superclass by sending the message addSuperPolicies to the newly created class. Note that these inherited policies are overridden by equivalently named policies that are explicitly assigned to the class using the message policyAt:put:.

Furthermore, a programmer can use the keyword super to refer to the superclass policy from within the definition of a policy that is assigned to the subclass using the message policyAt:put:. Note that super always refers to the superclass policy with the name of the newly added policy (*i.e.*, the first argument to the message policyAt:put:).

As an example, consider the class Morph and its subclass SketchMorph described in section 2.2. Since SketchMorph needs to override only the two drawing methods, it uses its superclass through the encapsulation policy **drawingHooks**. However, it still offers all the encapsulation policies specified by Morph so that it does not unnecessarily restrict its clients. This is done by using the message addSuperPolicies. In addition, SketchMorph overrides the inherited policy drawingHooks so that it also contains the method selector drawPDF:.

```
(Object subclass: Morph)
    instanceVariableNames: 'submorphs owner color bounds';
    policyAt: basicUse put: PMorph;
    policyAt: basicExtend put: PAll;
    policyAt: drawingHooks put: PAll noOverride + [drawOn: drawPostscriptOn:];
    policyAt: symbsubmorphManagement put: PAll noOverride + [addFront: addBack: ...]

(Morph subclass: SketchMorph withPolicy: drawingHooks)
    instanceVariableNames: 'form';
    addSuperPolicies;
    policyAt: drawingHooks put: super + [drawPDF:]
```

4.6 Customizing Encapsulation Policies

An important feature of encapsulation policies is that the client is not only allowed to select a policy offered by a class but can also customize the selected policy according to its needs. In our implementation, this is done by sending the messages −, ∗, and noOverride to the symbol corresponding the selected policy. As an example, assume that the class Point described in section 2.3 is defined as follows:

```
(Object subclass: Point)
    instanceVariableNames: 'x y';
    policyAt: basicUse put: [x y moveTo: radius degrees dotProduct: = ...];
    policyAt: basicExtend put: PAll
```

Although both encapsulation policies offered by this class declare the method = as overridable, we can still define a subclass ColoredPoint where implementing the method = does not override the superclass method. We do this by first selecting the policy **basicUse** and then customizing it so that it allows the method = to be called but not to be overridden.

```
((Point subclass: ColoredPoint withPolicy: basicUse - [↓=])
    instanceVariableNames: 'rgb';
    addSuperPolicies
```

Note that in our implementation, the programmer of the subclass decides whether the method = should override or simply reimplement the superclass method by means of specifying the encapsulation policy: the method = implemented in the subclass overrides the superclass method if and only if the encapsulation policy allows it. This means that the encapsulation policy not only specifies whether a method *can* override the superclass version but also whether it *will* override the superclass version.

5 Evaluation and Discussion

In section 2 we identified a set of limitations that are caused by the encapsulation models of state of the art object-oriented programming languages. In the following, we present a point-by-point evaluation of how composable encapsulation policies solve these problems in a simple and elegant way. Furthermore, we briefly discuss additional constraints for defining encapsulation policies in subclasses and compare encapsulation policies to Java interfaces.

5.1 Access Rights are Inseparable from Classes

Encapsulation policies are separate and independent from the implementation of a class. This allows us to express encapsulation policies in a reusable way and share them between arbitrary classes. Since these policies are not only sharable but also composable, it is possible to define new policies by combining, modifying and extending existing policies.

This significantly raises the level of abstraction because a programmer does not have to explicitly associate encapsulation attributes with the implementation of every single method. Furthermore, it reduces implementation and maintenance overhead because encapsulation decisions do not have to be duplicated in the first place and are therefore much easier to adapt if necessary.

Illustration. Let us reconsider the example of section 2.1. When we implement the classes Collection and Path with this approach, we do not have to assign any encapsulation attributes to the implementation of their methods. Instead, we can create a named encapsulation policy PCollection that contains all the accessible collection methods as well as the corresponding access rights and then use it as an encapsulation policy for both Collection and Path. In our Smalltalk implementation, this could be done as shown in section 4.4.

Besides the fact that we do not have to duplicate the encapsulation decisions for the collection protocol, this example illustrates also other advantages of encapsulation policies:

- Since encapsulation policies specify only accessible methods, the classes Path and Collection can use different internal method names (*e.g.*, internalAt: *vs.* basicAt: and unsafeAdd: *vs.* privateAdd:) and still use the same encapsulation policy. Furthermore, a programmer can add, remove or rename such internal methods in either class without having to change the encapsulation policy.
- The encapsulation policy PCollection can also be shared if a certain method, for example removeAll:, should only be accessible for clients of Collection but not of Path. This can be done by using the policy PCollection – [removeAll:] in Path.
- The encapsulation policy PCollection can be used in any class that provides the collection protocol. This means that independent of how such a class is implemented, the programmer does not have to deal with encapsulation on a per-method level and can instead just reuse the policy PCollection.

This stands in contrast to the interface-based approaches of languages like Java and C#, which do not support reuse of encapsulation decisions even if multiple classes implement the same interface (see section 5.5 and the discussion of C#'s CAS in section 6).

5.2 Client Categories are Fixed

We avoid this problem by allowing the designer to specify an arbitrary number of independent encapsulation policies for a given class. This allows a designer to implement a class with multiple usage scenarios in mind and to explicitly *specify* and *document* this by giving each of these scenarios a descriptive name and assigning it to an encapsulation policy. Another programmer can immediately see which usage scenarios a given class supports and then select the encapsulation policy corresponding to the usage scenario that is most appropriate.

In contrast to existing approaches, the designer can specify these encapsulation policies in a way that is independent of a particular mode of use. This raises the level of abstraction because it allows the programmer to think in a conceptual rather than an operational way. It is based on the realization that as long as it is not possible to sidestep such a conceptual policy, it is not relevant for the designer of a class whether a client uses the class by inheritance, instantiation, or any other mode of use such as automated delegation.

The fact that a particular encapsulation policy can be applied for both inheritance and instantiation gives a client the freedom to choose the mode of use that is most appropriate for its needs. In particular, it avoids those situations where a programmer is forced to inherit from a class just because this is the only way to obtain certain access rights, even if this is from a design point of view not appropriate (*e.g.*, when Stack inherits from OrderedCollection just to be able to access some internal methods of the collection) or not possible (*e.g.*, in a language that does not offer multiple inheritance).

Illustration. With our approach the limitation of a fixed set of categories can be avoided by associating different encapsulation policies with the class Morph. In our Smalltalk implementation, the class Morph and its subclass SketchMorph could be defined as shown in section section 4.5.

Another scenario in which it is useful to be able to assign multiple encapsulation policies to a class is when changes to the implementation of a class should be accessible for new clients without breaking existing ones. As an example, suppose that a vendor of a graphics framework ships a class GraphicalObject that is extensively subclassed by its customers. At a later point, the vendor would like to add the capability of *alpha-blending* to this class and therefore needs to add a few more internal methods such as transformAlpha:.

With traditional encapsulation approaches, these methods would be declared as protected since it should be possible to override them in new subclasses. However, doing this can break existing subclasses [SLMD96] because they may have introduced the same method name for another purpose! In our model, this dilemma can be solved by leaving the existing encapsulation policies as they are

and instead assigning the class a new encapsulation policy (*e.g.*, under the name withAlphaBlending) that contains these new methods. As a consequence, the new methods are completely invisible to all the existing subclasses that use the class through an old policy, whereas they are available for new clients that want to take advantage of them.

5.3 Access Rights are not Customizable

We allow a client not only to select the most appropriate reuse policy, but also to customize this policy so that it best matches its individual needs as long as it does not violate the restrictions defined by the designer of the class.

This allows one to reuse a class in a way that may not have been anticipated by the designer. Furthermore, it allows a client to specify a customized encapsulation policy that contains only the access rights that are effectively necessary, and it therefore *minimizes* the interdependencies between the class and its client. Minimizing these interdependencies is important because each access right that is granted by a policy comes together with a *risk* (*e.g.*, to accidentally and inappropriately call or override a method), *restricts the freedom* of the client (*e.g.*, in Java, a subclass cannot use the name of a protected superclass method for a method that represents a different concept), and makes the code more *fragile* with respect to changes in the used class (*e.g.*, unintended name clashes that can occur when a new internal method is added to the used class).

Illustration. In section 4.6, we have already shown how customizing encapsulation policies allow a programmer to solve the problem introduced in section 2.3, even if the designer of the class Point did not anticipate a client such as ColoredPoint that needs to implement a method == that is not compatible to the original implementation.

As another illustration, consider the class Morph introduced in section 2.2 and assume that its designer only provided the encapsulation policy PAll, which exposes the complete interface to its clients. Now suppose that another programmer would like to make a subclass TurtleMorph that overrides the method drawOn: and implements the turtle-specific methods go: and pointNorth using the methods position: and rotate:. To minimize the interdependencies to Morph, the programmer of TurtleMorph can still access Morph through a minimal policy that grants only the rights that are needed. This is done by using the intersection of the policy basicExtend and the policy that only allows the selectors position: and rotate: to be called and gives full access to the selector drawOn:.

```
((Morph subclass: TurtleMorph withPolicy: basicExtend * [↑position: ↑rotate: drawOn:])
    instanceVariables: '';
    ...
```

5.4 Constraints for Encapsulation Policies in Subclasses

In section 3.4, we formally stated the constraints that encapsulation policies impose on the clients of a class, and we have pointed out that these constraints

guarantee only that the encapsulation restrictions expressed by the designer of a class can never be violated in a direct or indirect client.

This has the advantage that a subclass can always assign *fewer* access rights to inherited methods, which is very useful in the dynamically typed language Smalltalk where implementation inheritance is a common practice [Tai96] and every method is traditionally fully accessible. For instance, it allows a programmer to create a class that can only be accessed through a restricted encapsulation policy even if the superclass (which may have been designed by another programmer) declares all the methods as fully accessible (*e.g.*, by using the policy PAll).

However, the price for this expressiveness is that it sacrifices substitutability of subclasses for superclasses. This means that if a superclass C offers a policy under the name p that grants full access rights to the method signature s, it may be that the policy that is offered by the subclass D under the same name does not grant any access rights for s. In fact, it may even be that the subclass D does not even provide a policy under the name p!

In a language like Java where subclassing implies subtyping, it may therefore be more appropriate to introduce additional constraints for the encapsulation policies offered by a subclass. For example, we could define that encapsulation policies are "inherited" and that a subclass cannot make these inherited policies stricter. This guarantees substitutability of subclasses for superclasses because every method signature that can be accessed through a policy named p in a class C can also be accessed through the policy p in all its subclasses.

Note that this additional constraint does not affect the ability to freely choose and customize a policy when creating a subclass nor does it prevent the designer of a subclass from offering additional policies (*i.e.*, policies that were not inherited from the superclass) that do not underly this constraint. Therefore, the programmer still enjoys all the conceptual benefits of encapsulation policies that are described in this paper.

5.5 Comparison to Java Interfaces

At a first glance, our notion of encapsulation policies resembles Java interfaces as they both specify a set of callable method signatures. However, aside from this structural resemblance, they are used for quite different purposes. Whereas the primary purpose of encapsulation policies is to express a usage contract between a component and its client, the purpose of Java interfaces is to declare subtype relationships in a way that is independent from subclassing.

As a consequence, Java interfaces are neither designed nor able to capture the encapsulation aspects of a class and separate them from the implementation. Instead, all the access attributes (*e.g.*, public, private, and protected) of methods are still declared together with their implementation. This means that regarding encapsulation, the information provided by interfaces is redundant because the definition of the methods in the class already defines all the encapsulation-related information.

Nevertheless, it may seem that Java interfaces offer a way for clients to reuse a class through different encapsulation policies by simply creating an instance

of the class and then using type casts to restrict access to this instance to an interface that is associated with the class. However, this sort of "policy" is not comparable to the one expressed with our approach because of the following limitations:

1. *Policies cannot be enforced.* Even if a client reuses a class C through an instance that has been type casted to the interface I, there is no way to enforce that this instance is not accessed through the complete interface defined by C. This is because it is always possible to use a downcast to convert the instance back to the type C.
2. *Policies can only be defined for one category of clients.* The policies defined by Java interfaces are only available to instances but not to subclasses. This means that all subclasses always have to reuse the class through the unchangeable policy that is defined by the access attributes in the implementation of the class.
3. *Policies cannot be defined independently.* The policies defined by Java interfaces are interdependent with the encapsulation decisions specified within the implementation of the class. This is because an interface can only be applied to a class that declares all the method specified by the interface as public, but this in turn makes it impossible to provide another policy that does not allow full access to these methods.

Realizing that Java interfaces are not expressive enough to be used as encapsulation policies, the other interesting question is whether encapsulation policies are expressive enough to be used as types, *i.e.*, whether encapsulation policies subsume interfaces. Even though we have neither formalized nor implemented a type system based on encapsulation policies, we strongly believe that this is possible.

Our belief that this is possible stems from the fact that encapsulation policies contain a superset of the information expressed by interfaces. In fact, both types of entities define a set of method signatures that are allowed to be called. The only difference is that encapsulation policies also capture all the other encapsulation aspects such as which of these methods are allowed to be overridden and reimplemented. Also the relationship of encapsulation policies and classes is quite similar to the relationship of interfaces and classes. In fact, a programmer can associate several possibly nested encapsulation policies to a class to express that the class conforms to the "interface" that is expressed by such a policy.

6 Related Work

We have already discussed the encapsulation mechanisms of the languages Java, C++, C#, and Eiffel in section 2. In this Section, we briefly discuss encapsulation mechanisms of other languages as well as some related research.

The encapsulation model of CLOS and Dylan follow the tradition of Lisp-based object-oriented languages such as Flavors (with the notable exception of CommonObjects [Sny86]). There is no direct access to slots as in Java or

Smalltalk. The access is always performed via accessors that can be generated automatically from the class description. However, it is always possible to access a slot value using the function slot-value[4]. There is no encapsulation of methods, which means that they are all public and late-bound.

Ada [Ame83] uses packages as a module system. These packages have a separated *definition* and *body*. Besides importing and exporting definitions, Ada 9X [Taf93] also allows sharing of packages. This is used for constructing hierarchical libraries, and solves the problem of private types only providing coarse control of visibility and the inability to extend packages without recompiling them. Hierarchical libraries are built by adding *child packages* to existing packages. The child packages can add definitions to their parent package and can see the internal body of their parent. Child packages can be made private, which means that they are only visible within the subtree of the hierarchy whose root is its parent. Moreover, within that tree, a private child package is not visible to the specifications of any non-private sibling (although it is visible to their bodies).

Ada also has protected types that are used for concurrent tasks. Protected types consist of a *specification*, where the access protocol is specified, and a *body*, where the implementation details are provided. Protected types contain a notion of visibility (clients can only use the procedures as defined in the access protocol), but they also control the access to the data these procedures work with: calls from clients to subprograms within a protected body are mutually exclusive.

Modula-3 is a statically typed, object-oriented language with single inheritance, and modules that consist of separate interface and implementation files. Modula-3 sports *partial revelation* [Fre95], which is a technique that allows inheriting from a class without making all its features visible. This is done by dividing class definitions across multiple files, each of them specifying a partial type for the class. Partial revelation allows a program unit to import only the relevant aspects of a class by selecting the corresponding type; the other aspects of the class are still available and can be revealed elsewhere in the program. Similar to encapsulation policies, this addresses the problem that different kinds of subclasses need to access a class in different ways.

However, there are many conceptual differences between the two approaches. For example, the Modula-3 approach does not model different access attributes: a feature is either visible (*i.e.*, fully accessible) or hidden; there is no mechanism for a more fine-grained distinction between allowing a feature to be called, overridden, and reimplemented. Furthermore, the Modula-3 types are not composable: it is only possible to use partial and full revelations to define a new type that reveals more features of an object type. Also, it is not possible for a subclass to customize the partial types that are available for a class.

[4] In CLOS, slot-value is in fact calling slot-value-using-class, which is an entry point of the MOP that allows controlling of slot accesses. Therefore it is possible to define a different encapsulation mechanism than the default one.

The object-oriented programming language Beta [MMPN93] is a block-scoped language where the visibility is given by the nesting of the blocks. There are no explicit visibility attributes that can be granted beyond that. However, Beta allows one to declare *virtual* patterns that can be extended in subpatterns. Furthermore, virtual patterns can be finalized (*i.e.*, made non-virtual) in a subpattern. For programming in the large, Beta also has a hierarchical module system to declare interface modules and implementation modules. The module system allows different implementation modules to be associated with the same interface module (so-called *variants*). Module visibility is also controlled by nesting.

C# and other .NET languages allow one to place CAS attributes on methods of interfaces and thereby reuse the constraint attributes across all classes implementing these interfaces. However, the *security* constraints that are expressed by CAS attributes and then checked at runtime by the .NET CLR are conceptually different from the *encapsulation* constraints that are expressed by our approach and C#'s access modifiers.

Wolczko also argues that existing class-based languages do not provide sufficient support for encapsulation [Wol92]. This is addressed by a Smalltalk-based research language called MUST, which offers additional features such as two types of self-sends and super-sends. This allows a programmer to express additional encapsulation issues in a very fine-grained way, by extending the language with additional mechanisms. In contrast, our approach is of a conceptual and language independent nature: it does not specify exactly what encapsulation issues (*i.e.*, access attributes) should be modelled, but it suggests to separate these encapsulation issues from the implementation and to make them first class.

The Jigsaw modularity framework, developed by Bracha in his doctoral dissertation [Bra92], defines module composition operators hide, show, and freeze to control how attributes of a module are encapsulated. Whereas hide eliminates the argument attributes from the interface of a module, show eliminates everything but the argument attributes from the interface of a module. The operator freeze allows one to control how attributes of a module are bound. It takes an attribute as an argument and produces a new module in which all references to the argument attribute are statically bound.

Altogether, these operators give a programmer fine-grained control over how a module should be encapsulated. Similar to our approach, they also allow the client of a module to decline access rights by hiding or statically binding attributes. However, the Jigsaw framework cannot capture such encapsulation decisions as separate, reusable entities, associate them to modules and apply them when a module is used.

Caesar's collaboration interfaces extend the concept of interfaces to include the declaration of expected methods, *i.e.*, the methods that a class must provide when bound to an interface [MO02]. However, they do not address the encapsulation problems that are addressed in this paper.

Sadeh and Ducasse present the introduction of *dynamic* interfaces in Smalltalk [SD02]. These interfaces represent a list of message selectors which are causally connected to the class that implements them. The system can be dynamically

queried to get the classes implementing a given interface. Dynamic interfaces can be derived from other interfaces or included in other interfaces. As Smalltalk is dynamically typed, dynamic interfaces mainly serve as documentation purpose. Contrary to encapsulation policies, dynamic interfaces do not deal with encapsulation aspects.

To avoid the fragile base class problem [MS98], researchers developed better ways to describe the contract between a class and its subclasses. Lamping proposes a limited specialization interface that expresses the calling relationships between the methods in the superclass [Lam93]. Reuse Contracts [SLMD96] bring the idea a step further by proposing a model in which the operations of class evolution are analyzed in the context of the calling dependencies in the superclass. Hence the evolution problems are categorized and detected with finer precision.

7 Conclusion and Future Work

In this paper we have proposed composable encapsulation policies as a way to improve the flexibility and expressiveness of object-oriented programming languages and to reduce the fragility of the resulting programs. Explicit encapsulation policies enable the expression of different usage scenarios for different classes of clients, they enable reuse of policies, and they enable client-specific customizations in a straightforward way.

We have outlined a general, language-independent model of encapsulation policies, and we have described a proof-of-concept prototype for Smalltalk that demonstrates the feasibility of the idea. Encapsulation policies can be incorporated into a language in such a way that there is an additional syntactic burden only when one wishes to make use of the feature. In other cases, default policies mimic the conventional approach offered by the language.

We are working on extending our proof-of-concept prototype to a full implementation of encapsulation policies in Smalltalk as well as Smalltalk with Traits. This will serve as the basis for a more detailed evaluation of the advantages of our approach in languages that feature non-standard composition mechanisms such as trait composition or automated delegation. Furthermore, we plan to investigate the impact of replacing the traditional encapsulation mechanisms of languages like Java with encapsulation policies. In particular, it seems that there could be interesting synergies if the notion of encapsulation policies would also be used as a type and could hence replace the notion of interfaces.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Tools and Techniques for Decomposing and Composing Software” (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and “RECAST: Evolution of Object-Oriented Applications” (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

References

[Ame83] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*, volume 155 of *LNCS*. Springer-Verlag, 1983.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[Fre95] Steve Freeman. Partial revelation and Modula-3. *Dr. Dobb's Journal*, 20(10):36–42, October 1995.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97*, pages 318–326. ACM Press, November 1997.

[Lam93] John Lamping. Typing the specialization interface. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 201–214, October 1993.

[MMPN93] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Reading, Mass., 1993.

[MO02] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA 2002*, pages 52–67, November 2002.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP'98*, number 1445 in Lecture Notes in Computer Science, pages 355–383, 1998.

[Nie89] Oscar 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.

[SD02] Benny Sadeh and Stéphane Ducasse. Adding dynamic interface to Smalltalk. *Journal of Object Technology*, 1(1), 2002.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 38–45, November 1986.

[Taf93] S. Tucker Taft. Ada 9x: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93*, volume 28, pages 127–143, October 1993.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[VRB00] John Viega, Paul Reynolds, and Reimer Behrends. Automating delegation in class-based languages. In *Proceedings of TOOLS 34'00*, pages 171–182, July 2000.

[Wol92] Mario Wolczko. Encapsulation, delegation and inheritance in object-oriented languages. *IEEE Software Engineering Journal*, 7(2):95–102, March 1992.