Open Surfaces for Controlled Visibility

Stéphane Ducasse, Nathanael Schaerli, and Roel Wuyts

Software Composition Group, University of Bern,
Bern, Switzerland
{ducasse, schaerli, wuyts}@iam.unibe.ch

Abstract. Current languages contain visibility mechanisms such as *private*, *protected*, or *public* to control who can see what. However, these visibility mechanisms are *fixed once for all*. Moreover, they do not solve all problems related to the visibility, and are typically of a static nature. In this position paper we present an open and uniform way of dealing with visibility and introduce *surfaces* : *i.e.,* list of methods that control the way the behavior of an object is accessible. We introduce two problems that other visibility mechanisms cannot solve, and show how surfaces can.
Right access, Visibility, Permissions, Implementation surface, Specification surface, Smalltalk

1 Visibility Mechanisms and their Problems

Several languages provide visibility mechanisms that allows classes to control which other classes can see (and send) their methods. For example, C++ [5], like Java [1], has *public*, *private*, and *protected* modifiers to control the visibility of methods. On top of these, C++ also provides the *friends* mechanism (which controls visibility among a group of classes that are not necessarily related in a single class hierarchy) and qualified inheritance. The advantage of such mechanisms is clear: it makes it possible for classes to control the visibility of the functionality they offer towards clients, promoting encapsulation.
However, we also several drawbacks in these approaches. For starters, the visibility mechanisms are purely used for static checking. It is for example not possible to promote a private message to become public at runtime. Moreover, there are also some problems statically. First of all, there is only a limited number of visibility modifiers (as said before, typically public, private and protected). Moreover, this set cannot be extended to suit particular usage scenarios.
To address these problems we propose *surfaces*. *Surfaces* are sets of selectors (method names). Classes can define one or more surfaces, and by default always have an `All` surface that includes all selectors for that class. Clients can, statically or at runtime, indicate through which surface they want to access an object. Last but not least, surfaces are first class objects.
Surfaces solve the problems that classic visibility mechanisms face since they are *open* (i.e., there is no restriction on the number of surfaces or over what they contain), and are applicable statically and at runtime.

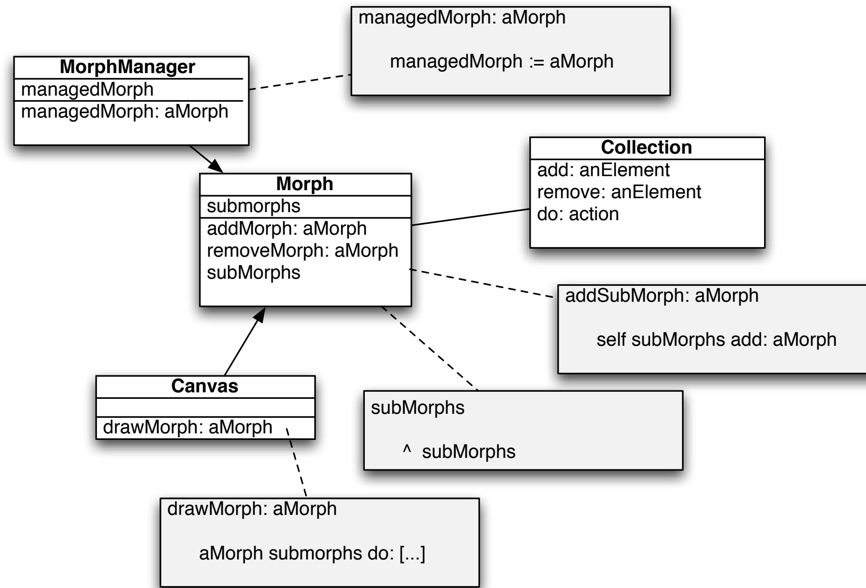

Fig. 1. The Squeak class `Morph`, containing an instance variable `submorphs` that contains a collection of Morphs, and a class `Canvas`, on which morphs can be drawn.

The rest of the paper is structured as follows. Section 2 discusses a motivating problem that gives a more detailed example about the shortcomings of current visibility mechanisms. Section 3 then introduces surfaces in detail. Section 4 talks about our motivations in designing the surfaces, and discusses some open issues. Section 5 concludes the paper.

2 Problem Illustration: Morph

The Squeak Smalltalk system includes a GUI paradigm called *Morphic* [2]. The basis of this system is formed by morphs, entities that combine model and view elements in a single place. This is implemented by the class `Morph`. Since morphs can have submorphs, this class contains an instance variable `submorphs`, that holds a collection of Morph objects. Morph has a number of methods to add, remove, and enumerate its submorphs. The idea is that subclasses and other clients have to use this protocol instead of directly manipulating the collection of morphs. Morph has two clients: the class `MorphManager` that only needs to use the methods `addMorph:` and `removeMorph:`, and the class `Canvas` that only needs to enumerate morphs (and not add or remove them). This setup is shown in Figure 1.

2.1 Problem 1

In order to force clients to use the methods `addMorph:` and `removeMorph:` whenever they need to add or remove submorphs, we want to make sure that the method `subMorphs` returns a read-only collection of submorphs. This problem is very hard to solve cleanly. We see three possible solutions, all with their specific drawbacks:

- the method `submorphs` of class `Morph` can return a copy of the collection. Depending on the situation, this solution can become very expensive, and is therefore not generally applicable.
- all the enumeration methods of the class `Collection` can be implemented on the class `Morph`. The methods dealing with enumeration can be made public, while the ones dealing with adding or removing can be made private. The big drawback of this approach is that a lot of methods have to be duplicated, which is not a good thing from the point of reuse and code duplication.
- we can create a subclass from the class `Collection` that defines the methods `add:` and `remove:` from Collection as being private. Then we can make the class `Morph` a friend of this new collection. Thus the class `Morph` can invoke the private methods because of the friend relationship, while other classes that get passed this collection can only invoke the enumeration method `do:`. However, although it solves the problem this solution is undesirable in a number of ways. First of all, a subclass has to be made just so that the access modifiers from collection can be changed. Even when we could destructively go in to modify the class `Collection`, there is still another problem. This is that the friends relationships that is needed is much too coarse grained. Suddenly, Morph has access to the complete private life of class `Collection`. Furthermore, it has access not only to the inner parts of the collection containing submorphs, but actually to any collection.

In general, we conclude that there is currently no generally applicable clean way to solve these kinds of problems. All the approaches have some severe drawbacks, employing tricks to address the problem. As we will see in Section 3, giving Collection two surfaces solves this problem cleanly.

2.2 Problem 2

A second problem is that the visibility modifiers that are required for the classes `MorphManager` and `Canvas` are opposite. As far as the class `Canvas` is concerned, the methods `addMorph:` and `removeMorph` would need to be private, while for the class `MorphManager` they need to be public. No C++ friends construct or Java package-based solution can solve this seemingly trivial problem.

3 Surfaces To The Rescue

This section defines what a surface is, how it solves the two problems from section 2, and then shows how the surface is actually described.
The following defines a surface:

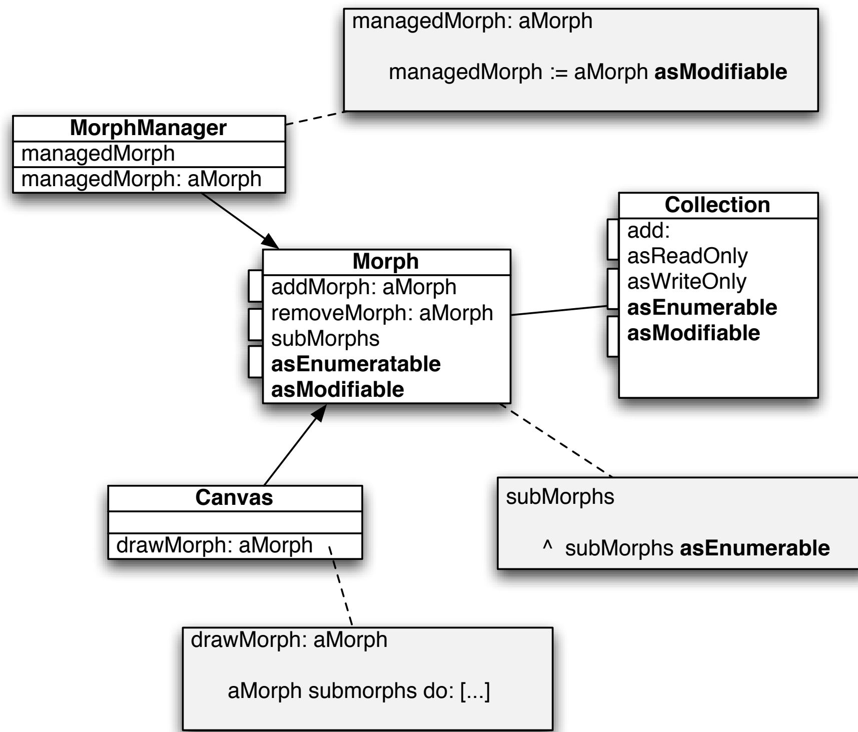

Fig. 2. The Squeak class `Morph`, containing an instance variable `submorphs` that contains a collection of Morphs, and a class `Canvas`, on which morphs can be drawn.

Definition: A surface is a list of method names that a client can use to invoke behavior of an object.
Once a surface is defined, objects can expose themselves through a surface. Clients that get passed such an object can only invoke the methods defined by the service. An object can restrict or change its visibility and can propose different surfaces to the same or to different clients. One of the surfaces can be assigned to be the default service (to be used when a client does not ask for a specific surface). By default, the `all` surface is used as the default, an implicit surface that returns all the selectors of the class.
The next sections have a detailed look at how the surfaces solve the problems mentioned in the previous section. Figure 2 shows the solution.

3.1 Solving Problem 1

Let's see how this solves the first problem described in Section 2.2.
First of all we define two surfaces on the class `Collection`: `enumerable` (containing selector `do:`) and `modifiable` (containing selectors `add:` and

`remove:`). Then we implement the following methods on the classes `Morph` and `Collection`, where the class Morph returns a read-only collection as a result of the message `subMorphs`. Note the new message `|`, which is used to expose an object through an interface (and which we'll discuss in more detail after this example).

```
Morph>>subMorphs
   "returns all the morphs owned by the receiver"
   ^ submorphs asModifiable


Morph>>addMorph: aMorph
   "add aMorph in the list of submorphs. Access instance variable directly."

   ^ subMorphs add: aMorph.


Collection>>asModifiable
   ''return myself exposed as a modifiable morph''

   ^ self | modifiable

Collection>>asEnumeratable
   ''return myself exposed as an enumerable morph''

   ^ self | enumerable
```

An important design point is that only the class which defines the surface can use it to expose its objects through it. So the receiver of the `|` message is always `self`, and can never be anything else. So the following expression is *not* allowed:

```
Morph>>subMorphs
   "returns all the morphs owned by the receiver"
   ^ submorphs | readOnly
```

Note that this point also applies to classes and their instances: Since an instance is another object than its class, a class cannot explicitly change the surface of an instance (of course it can change it by sending a message such as "`asReadonly`" if there is one). However, this means that the methods that explicitly change the surface must be implemented on the instance side.

In short, this example shows that surfaces provide a mechanism that allow a class to expose groups of methods under a certain name. Moreover, the mechanism is open so that new surfaces can be added. What we haven't shown explicitly is that different instances of the same class could see the same object with different surfaces, since the services are not applied on classes but on objects.

3.2 Solving Problem 2

To solve the problem described in Section 2.2 we give the class `Morph` two surfaces, analogously to the interfaces on the class `Collection`: `default` (containing the selector `submorphs:`) and `modifiable` (containing the selectors `addMorph:` and `removeMorph:`). Moreover, we set the `default` surface to be the default one.
We then implement the following two methods on the classes `Canvas` and `MorphManager`

```
Canvas>>drawMorph: aMorph
    ''Draw a morph on the canvas''

    ...
    aMorph submorphs do: [:eachSubMorph | self drawMorph: eachSubMorph].

MorphManager>>manageMorph: aMorph
   ''Mutator method to set the morph that will be managed''

   manageMorph := aMorph asModifiable.
```

In short, this example shows that surfaces support parametrised usage, where the client can choose between different options that are offered.

3.3 Describing Surfaces Declaratively

At first we though of describing a surface as an explicit collection returning selector names. However, this would mean a lot of overhead maintaining this set. Suppose in our example that we add another enumeration method `collect:` on the class `Collection`. Then we have to remember to update the surfaces that we want this method to appear in. This would make the system cumbersome to use, even with proper browser support. Therefore we propose to tag the methods themselves with the surfaces where they want to occur. The surface definition itself then simply describes which tag it used. So, in the previous example, the methods `add:`, `remove:` and `do:` would look as follows (we only show the tages for the methods):

```
Collection>>add: anObject
   <#modifiable>
   ...

Collection>>remove: anObject
   <#modifiable>
   ...
```

```
Collection>>do: action
   <#enumerable>
   ...
```

Then a surface definition could simply assemble all methods with a certain tag. We are undecided yet if we want to do this explicitly in the class definition, or leave it implicitly. Making it implicit, where one just have to tag methods, and the tags can then automatically be used as surfaces, has the advantage of being simpler. However, it has the disadvantage that simple spelling mistakes result in the unwanted definition of new surfaces. Moreover, an explicit definition mechanism would allow composition of surfaces. For example, we could define the surface `modifiable` to be the surface `all` minus the surface `enumerable`:

```
Object subclass: #Collection
   instanceVariables: 'a b'
   classVariables: ''
   poolDictionaries: ''
   surfaces:
      #enumerable -> #(do: ... ).
      #modifiable -> #ALL - enumerable.
```

4 Discussion

Mutual Contract. Every class defines the surfaces that clients can use, and a default one that is used when the client does not specify anything specific. Moreover, every class automatically has a surface `all`, that contains all the selectors for that class. Therefore any class that is not happy with the provided surfaces can use that surface to still access everything. This makes the surfaces fit with the Smalltalk environment that we envisage, since in Smalltalk instance variables are private but all methods are public. However we agree that this is a negotiable feature (strong encapsulation versus strong encapsulation with an overriding mechanism), but omitting the `all` interface makes the system strong, even though the client can still choose which surface it wants to use. This was shown in the second example in Section 3.2.

About the Notion of Identity in Presence of Surface The question of identity in presence of surface has to be addressed. Indeed is a reference to an object the same For example, is submorphs == self subMorphs true? We propose to have different methods to check if the references to the same object are made via the same surface.

Implementation There is currently no implementation that supports surfaces. We have a very good idea on how to add surfaces to the Squeak

Smalltalk system [2], and we hope that by the time of the workshop we will have something to show. The implementation boils down to doing a clever management of method dictionaries, a technique that was already employed for the aliasing mechanism as found in the Traits system [4]. As such it should constitute only a minor overhead in space, and not in time. Of course, next revisions of this paper will go more in detail once the system is implemented and working.

Related Work The most relevant related work that we found is in the ACE and JAC projects [3]. In these projects, aliasing of objects wants to be supported in an encapsulated way. As such it addresses the issues raised by our first problem in Section 2: how to share the collection of submorphs, but control who can modify it. The main concept presented is "readonly types": through a reference of a readonly type it is not possible to change any part of the transitive state of the referenced object.
The main difference with surfaces is that surfaces are open, and allow one to add visibility schemes when needed. We envision a system where every object has several interfaces, that are not only used to control aliasing but also to control other aspects of relevance to that application. We propose surfaces as the foundation for such systems.

5 Conclusion

This paper proposes surfaces, a mechanism to control visibility in object-oriented programming languages. The distinctive features of surfaces are that it is open (making it easy to specify visibility rules, and they are not limited to a set of built-in ones), and that they are applied on objects.

References

1. K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1996.
2. M. Guzdial. *Squeak - Object Oriented Design with Multimedia Applications.* Prentice-Hall, 2001.
3. G. Kniesel and D. Theisen. Jac - acces right based encapsulation for java. *Software: Practice and Experience (Special issue on Aliasing in object-oriented systems)*, 31(6), 2001.
4. N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003*, LNCS. Springer Verlag, July 2003.
5. B. Stroustrup. The C++ programming language – reference manual. Computing Science Technical Report 108, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, Jan. 1984.