Protected applications in the Smalltalk-80 System

L. Peter Deutsch
Xerox, ParcPlace Systems

This is a working document proposing mechanisms for better encapsulation and protection of independent applications for the Smalltalk-80 language and system.

Comments are solicited. For changes in successive versions, consult the version history.

Version history:

[2] 10 May 1987: split into three documents (scopes, pkg, and load); incorporated comments by Richard Steiger, Mark Miller, and George Bosworth; decoupled this document from the standards activity; added new section on debugging and errors; added syntax for creating private security area and name scope.

[1] 25 February 1987: first version (pkg87), distributed to implementors at the Feb. 26-27 workshop.

Introduction

Conventional language systems support the development of applications that can be made available in object code form only, and are thereby protected from unwanted interference or scrutiny (subject to the ability of the programmer to create pointers to arbitrary machine locations). In contrast, the current Smalltalk-80 system is completely open: all objects (including source code) are available for inspection and modification, and anyone can send any message to any object to which the sender has a reference. This greatly reduces the incentive to design in a strongly modular way, and tends to lead to a tangled, monolithic system. In order to alleviate this problem, and also to encourage the development of third-party software, this document proposes a mechanism for truly encapsulated applications within the Smalltalk-80 system.

In theory, an object-oriented language such as Smalltalk provides excellent abstraction, since messages are interpreted relative to the class, and inter-program protection, since there is no way to violate the abstraction boundary of an object. In practice, there are several barriers to meeting this promise:

- All class names are global, so any object can send a message to any class. This makes protection awkward, because typical applications have both public and private classes.
- Any object can send any message to any other object it has a reference to. This makes protection awkward, because typically objects

have both public and private protocols.
- Debuggers need a controlled way of breaching the abstraction boundary, but in the current system there is no way to restrict access to messages such as instVarAt:.

We propose to address these problems by a combination of language and system facilities. A separate document describes new mechanisms for more flexible control of name visibility, which addresses the first problem mentioned above. Another document (in preparation) describes a proposal, largely due to Mark Miller, for a systematic approach to protecting objects from direct inspection other than by authorized subsystems such as debuggers. Consequently, this document only deals with the second problem: how to provide encapsulation without having to add new mechanisms to restrict the ability of objects to send arbitrary messages to arbitrary other objects.

Our solution to the encapsulation problem is based on separating public and private protocols into different classes. This approach has significant overhead, and we believe it is most appropriate for applications with stringent information hiding needs. We believe that a superior solution would be to allow message selectors to be more general objects, not necessarily literal Symbols, which would allow a package to have selectors that cannot be named by its clients or anyone else. However, this approach appears to require more new machinery than the one we propose here, as well as being less flexible in some ways (e.g. it doesn't directly provide for dynamic revocation of capability).

The proposed approach involves no changes to any existing part of the Smalltalk language or system; however, it depends on the facilities for flexible name scoping and meta-level protection.

Meta-level facilities

Mark Miller's proposal includes an explicit notion of a "security area", an object that controls access to the representation of other objects. We propose that this notion be implemented in a way that allows private application classes and their instances to belong to security areas different from the normal "open" area, and to different areas for different applications. We only sketch some of the ideas here: for more information on this concept, read Mark's proposal.

For private messages to have any value as a protection mechanism, they must be protected in the same way as an object's implementation. (This is, for example, consistent with the Actors view that objects are individually fully responsible for decoding their messages.) For this reason, the operations that allow direct access to an object's class (including the method dictionary) are also placed in the "meta" category.

Since Mark's proposals are complex and far-reaching, we only discuss a subset of them here. This subset is aimed specifically at the following problems:
- Controlling access to an object's storage representation (the functionality of instVarAt:, instVarAt:put:, become:, class).
- Controlling access to the state of a class (method

dictionary, class variables, superclass reference).

As in Mark's proposal, we introduce the concept of a MetaObject. A MetaObject holds a reference to an object (called the 'subject' of the MetaObject), and provides direct access to the instance variables of the subject. Access is controlled by controlling the creation of MetaObjects for a given subject. Rather than provide instVarAt: and instVarAt:put:, we provide a message

<InstVarDictionary> := <MetaObject> instVarDictionary

that returns an object that provides Dictionary protocol to actually access the instance variables of the MetaObject's subject by name. (We do not specify how this occurs: presumably the InstVarDictionary invokes protected messages of the MetaObject, that in turn call primitives similar to the present instVarAt:[put].)

To control the creation of MetaObjects, we introduce the notion of a SecurityArea, again as in Mark's proposal. Only a SecurityArea can create a MetaObject. The Smalltalk system starts out with a single SecurityArea, which can create a MetaObject for any existing object. Ideally, the primitives for creating objects should specify in what SecurityArea the object is to be created. However, since we have had no experience with the SecurityArea concept, we are unwilling to require implementors to provide this capability, which may add significant complexity to the memory manager. Instead, we adopt a much more limited position that addresses some, but not all, the security needs of protected applications: we divide objects among SecurityAreas on a per-class basis, and propose that each class specify which SecurityArea can access it and its instances, i.e.

<SecurityArea> := <Behavior> securityArea.

Protected interfaces

As mentioned above, and explained in more detail below, applications can effectively control what objects are available to their clients by name. In this section, we describe how they can use this facility to provide a completely protected interface to clients. No new language or system facilities are involved: this is purely a matter of convention. At the end of this section, we provide a detailed example.

If an application wishes to present a protected interface, it is not sufficient to only make a particular class or classes visible:

- The client can instantiate the class in an uncontrolled way.
- The client can send any message to instances of the class, not just those intended for public use.
- The client can do other operations on the class, such as changing its methods.

To prevent clients from instantiating an application class, we propose that the application not make classes available to the client. Instead, the application should export a "factory" object that only understands a few object-creation messages. This factory, having been compiled in the naming environment of the application, can refer to and instantiate

application classes as needed. As we will see below, factories are sufficiently stylized that they can be generated automatically from declarations of what messages should be public.

To prevent clients from sending private messages, we propose that the objects provided dynamically by the application to the client not be instances of the application classes that actually do the work. Instead, the application should provide interface objects that only understand the public messages. The interface object should hold, in an instance variable, a reference to an object that actually provides the functionality, and forward the public messages to it. Interface objects obviously need an initialization message to set this reference, and this message must be protected in some way so that clients cannot change the reference once it has been set. For this message, we propose a different technique, based on checking a "key" against an object not visible to clients: see the example below for details. Again, most of the code for interface classes can be generated automatically from declarations.

It has been suggested we do not actually need to use a key for validating the initialization message for the interface object: we can simply check whether the instance has been initialized yet (i.e. the instance variable holding the real object is non-nil), and signal an error if this is the case. This approach seems to work for the simple case presented below, but may not be adequate for more complex situations.

As discussed in the section on meta-level facilities above, by placing an application in a different security area, we can prevent clients from accessing its machinery (such as the method dictionaries of its classes, or the instance variables of its instances) in an uncontrolled way.

Here we give a very simple example of a protected application: a Counter that is initially zero, and can only be incremented and read. Even in this simple example, an interface class is required, because the initialization message to the Counter must be protected. None of the three classes exhibited below are visible to clients: only the static variable named CounterMaker is exported. The static variables (classes) CounterFactory, CounterInterface, and Counter exist only in a scope private to the application, as does the variable CounterKey.

The code given here for creating security areas and name scopes is completely speculative, and may not be at all appropriate in general. Indeed, the syntax is not quite acceptable, since it includes temporary variables declared in the middle of a statement sequence, and a mixture of language syntax and file "chunk" delimiters.

```
" ------ Create the SecurityArea ------ "

| myArea |
myArea := SecurityArea current newArea.
myArea control: [

" ------ Create the private name scope ------ "

| myGlobals |
```

```
myGlobals := Dictionary new.
globals at: #CounterArea put: area.
globals at: #CounterGlobals put: globals.
globals enclose: '

" ****** ALL THE 'S SHOULD BE DOUBLED FROM HERE ON ****** "

" ------ The factory class ------ "

CounterFactory := Behavior
    newSuperclass: Object
    instanceVariables: #()
    classVariables: #()
    poolDictionaries: #().

!CounterFactory methods forCategory: 'Counter creation'!
new
    "Make a new Counter for the client"
    ^CounterInterface new initialize: CounterKey! !

" ------ The interface class ------ "

CounterInterface := Behavior
    newSuperclass: Object
    instanceVariables: #(#counter)
    classVariables: #()
    poolDictionaries: #().

!CounterInterface methods forCategory: 'Counter creation'!
initialize: aKey
    CounterKey == aKey
        ifFalse: [self error: 'Unauthorized message'].
    counter := Counter new.
    counter initialize.
    ^self! !

!CounterInterface methods forCategory: 'client accessing'!
increment
    counter increment!
value
    ^counter value! !

" ------ The real counter class ------ "

Counter := Behavior
    newSuperclass: Object
    instanceVariables: #(#count)
    classVariables: #()
    poolDictionaries: #().

!Counter methods forCategory: 'accessing'!
initialize
    count := 0!
increment
    count := count + 1!
value
```

```
        ^count! !

    "Create the key.  This can be any mutable object whatever.  It
    can't be an immutable object, since we can't be guaranteed that
    two immutable objects with the same contents won't be ==."

    CounterKey := Array with #Counter "a mutable array"!

    "Create the CounterMaker.  CounterMaker is the only static
    variable defined here that is visible to clients."

    CounterMaker := CounterFactory new!

    Smalltalk at: #CounterMaker put: CounterMaker!

    " ------ End of scope of CounterGlobals ------ "

        '.

    " ------ End of scope of CounterArea ------ "

        ]!
```

Debugging and errors

If an error occurs inside a protected application, we must ensure that the debugger does not allow us to violate a protection boundary. For example, Contexts probably need to be created in the SecurityArea of their receiver. The debugger probably should not even show Contexts in SecurityAreas to which the user does not have access. A complete discussion of this problem is beyond the scope of this document.