Class Naming and Privacy in Smalltalk

A single remedy for two design and reuse problems

Nik Boyd

Smalltalk lacks mechanisms for defining private classes and private methods. Without private classes, class naming conflicts can occur. Without private methods, encapsulation suffers. While global name spaces can help resolve class naming conflicts, first-class subsystems with private classes can resolve both problems.

Once a Smalltalk developer has learned the essentials of object-oriented design and programming, new issues regarding object system design and systematic reuse begin to surface. There are two factors that contribute to design and reuse problems in Smalltalk—both related to visibility: 1) all classes are visible to (and usable by) all other classes—Smalltalk has a single global name space; and 2) all the methods of a class are visible to (and usable by) its clients, in spite of the fact that some of its methods may be intended for the private use of the class.

This article considers these issues and proposes a single remedy for both problems, without changing the Smalltalk language and with relatively minor changes to the development environment.

The Name Space Problem

Because classes are globals in Smalltalk, they are visible to all other classes (as well as the programmer). This visibility is excessive. It can contribute to information overload for novice (as well as experienced) Smalltalk programmers. It can also cause class-naming conflicts when teams of developers integrate class libraries that have been developed separately.

Few Smalltalk environments include facilities and tools for integrating and organizing large libraries of Smalltalk classes, though some environments and third-party products provide tools for organizing Smalltalk source code into more manageable units (e.g., packages, applications, and the like). While these facilities help to reduce the number of classes immediately visible to a programmer, they do not eliminate the problem of having a single name space.

Adding prefixes to class names has become the common practice for dealing with this limitation. This practice helps prevent conflicts for commonly used names. However, while class-name prefixes prevent potential name conflicts, they degrade the readability and understandability of the class names. Understanding degrades further when the prefixes abbreviate what should be meaningful subsystem names.

Previous Approaches to Name Space Partitioning

There are only a few workable approaches to partitioning the Smalltalk class name space. Modular Smalltalk[1] addressed some of the issues related to the name-space problem. However, Modular Smalltalk redefined some of the fundamental characteristics of the language and its development environment.

While clear benefits can be gained from a static version of the Smalltalk language—that is, better performance at runtime—it remains unclear whether some of the existing benefits of Smalltalk might be sacrificed (such as dynamic and rapid application development). Perhaps the best of both worlds can be integrated into some future Smalltalk environments. Smalltalk MT looks promising in this regard.

Meanwhile, other commercial Smalltalk implementations still show the early origins of Smalltalk as an interpreted language. Objects and classes are defined and constructed dynamically using messages. These messages are compiled and evaluated in the context of an image-based object memory.

Global Behavior Pools vs. First-Class Subsystems

Another proposal for dealing with the name space problem was explored.[2] This proposal used global pool dictionaries to supplement the name space provided by the System Dictionary. The article showed how pool dictionaries (which usually contain system constants) could be extended to hold classes. Then, other classes could subscribe to these behavior pools and use the classes defined in such pools—by including the pool names in the poolDictionaries: portion of their class definitions.

While this proposal does help partition the class name space, the pools themselves are still global, and the classes defined in such pools are still public in the sense that

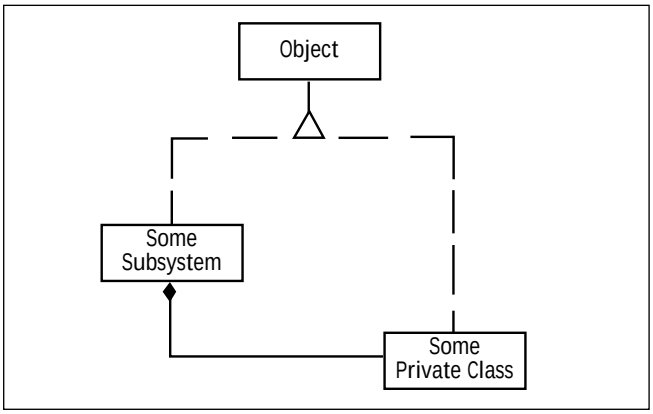

Figure 1. A Subsystem aggregates a Private Class.

they are still global. Such a pool-based facility is similar to C++ namespaces and the hierarchical namespaces provided in QKS' SmalltalkAgents. They provide separate domains for class names without considering whether those classes might need to be encapsulated in an object system design.

In contrast, first-class subsystems support the definition of truly private behaviors. Thus, subsystem classes support the organization and encapsulation of clusters of related classes. This facility is similar to the C++ nested class. Private classes are nested within the scope of a subsystem class, which serves as the public interface for the private classes it contains. A subsystem class can provide or restrict access to the private classes it contains, based on the needs of the subsystem design.

As in C++, global behavior pools and subsystems can complement each other for resolving class name conflicts, and for providing behavior encapsulation. However, the remainder of this article focuses on the benefits of subsystems and support for their implementation in Smalltalk.

The foundations for implementing first-class subsystems in Smalltalk were established in the February 1993 issue of *The Smalltalk Report*, [3] in which subsystems were called "modules." The terms "subsystem" and "private class" serve to better describe the intentions underlying this technology, and will be used throughout the remainder of this article.

SUBSYSTEMS AND PRIVATE CLASSES

A Modeling Notation for Private Classes

Subsytems are useful for partitioning the behavior of object systems. Objects are nothing less than small systems, and systems are nothing more than large objects.

There is no conceptual difference between the responsibilities of a class, a subsystem of classes, and even an application; it is simply a matter of scale, and the amount of richness and detail in your model.[4]

For these reasons, it would be convenient to have an object model notation that shows the relationship between objects and systems. The notations that have been proposed previously have been internal rather than external (for example, they depict subsystems by nesting entities graphically). Such internal notations do not scale well graphically when they are applied to the design of large systems, especially when subsystems are nested (for example, when a subsystem class contains a private subsystem class).

First-class subsystems fully contain their private classes, including their definitions. Thus, private classes can be said to be parts of their (public) subsystem class (they are aggregated at the meta level). For this reason, the relationship between a private class and its containing subsystem class will be depicted using a variant of the OMT notation for aggregation.

In Figure 1, Some Subsystem is a first-class subsystem within which Some Private Class is defined. The object models used in further discussions will depict such meta-level aggregations of private classes using a special diamond (as shown in Figure 1).

Defining Subsystems and Private Classes

In addition to introducing a new graphical notation for depicting the design relationship between subsystems and private classes, this article introduces new message formats for defining subsystem classes and their private classes in Smalltalk.

```
"Define a new subsystem."
SomeSuperclass
subsystem: #SomeSubsystem
```

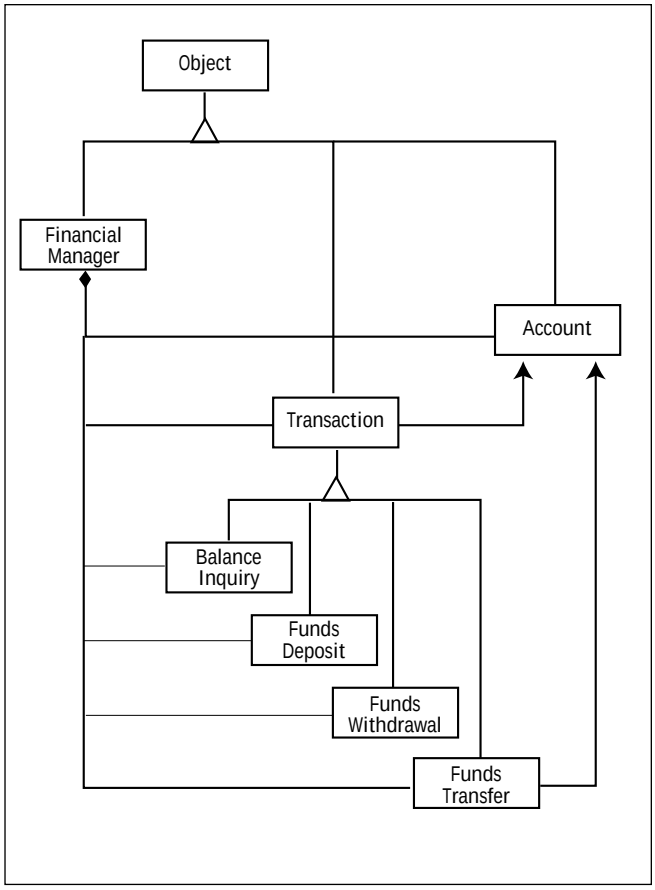

Figure 2. Financial Management Subsystem.

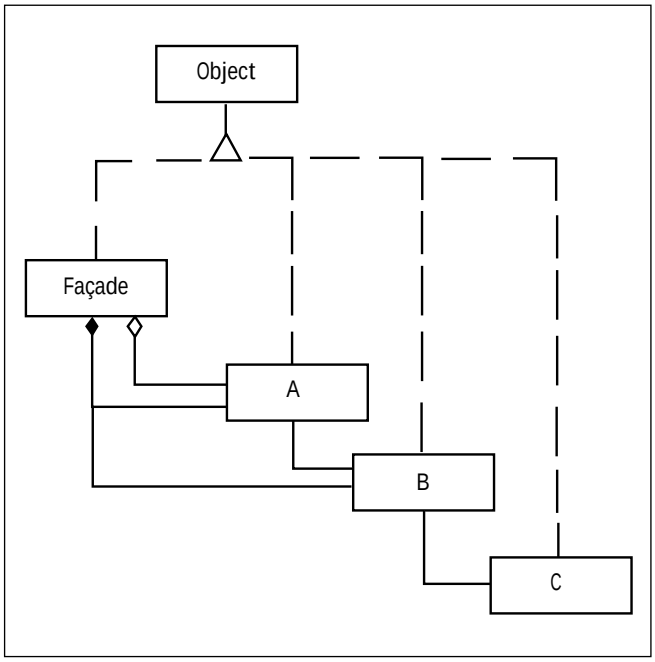

Figure 3. Object Model for a Façade.

```
instanceVariableNames: '...'
classVariableNames: '...'
poolDictionaries: '...' !
```

Notice that subsystem classes support instance variables, class variables, and pool dictionaries, just like ordinary classes. Subsystems are first-class objects. A subsystem class is just like any other class, except that its classPool may contain private classes in addition to the usual class variables. These details will be discussed further in the Implementation section.

Hereafter, in order to simplify the class definitions, the message portions after the subclass name will be elided. However, please remember that the entire list of message arguments is intended and supported for all such abbreviated class definitions. For example, the following partial message shows the abbreviated form of a subsystem definition.

```
"Define a new subsystem."
SomeSuperclass
subsystem: #SomeSubsystem ... !
```

A private class can begin the lineage of a private class hierarchy within a subsystem. Such a private base class must be defined differently from the other private classes derived from it. In particular, such a private base class must identify not only its superclass, but also the subsystem that contains it. The following partial message shows the abbreviated form of such a private base class definition.

```
"Define a private base class."
AnotherSuperclass
subclass: #SomePrivateClass
in: SomeSubsystem ... !
```

Of course, private classes can also be private subsystem classes.

```
"Define a private subsystem class."
AnotherSuperclass
subsystem: #SomePrivateSubsystem
in: SomeSubsystem ... !
```

Once a private class hierarchy has been introduced in a subsystem, the private base class and all the subsequently derived private subclasses can be located relative to the base of the private class hierarchy. The following partial message shows the abbreviated form for defining derived private classes.

```
"Define a private subclass."
SomeSubsystem @ #SomePrivateClass
subclass: #SomePrivateSubclass ... !

"Define a private subsystem."
SomeSubsystem @ #SomePrivateClass
subsystem: #SomePrivateSubsystem ... !
```

Visibility Rules for Classes and Subsystems

The visibility and scoping rules for private classes are similar to those found in C++ for nested classes. The classes defined outside a subsystem are visible to the private classes defined inside a subsystem, while the private classes defined inside a subsystem are not (immediately) visible to the classes defined outside a subsystem. Also, the classes defined within a given scope of visibility are visible to each other. Thus, the private classes defined inside a subsystem are visible to each other, just as the classes defined in the System Dictionary are visible to each other.

Classes defined outside a subsystem may be used directly by name in the methods of classes inside a subsystem. Class names are resolved by looking first in the local scope, and then progressing outward through the enclosing scopes until the named class is found.

The binary message @ serves a role similar to that of the scope resolution operator :: in C++. It can be used to locate a private class relative to its enclosing scope(s). Compare the following Smalltalk and C++ expressions:

```
Smalltalk    ( SampleManager @ #SamplePrivateClass )
C++          ( SampleManager :: SamplePrivateClass )
```

DESIGNING OBJECT-ORIENTED SOFTWARE SYSTEMS

Subsystems provide a coherent way to design and organize Smalltalk classes that collaborate closely. Several examples of subsystem designs are included in the book, DESIGNING OBJECT-ORIENTED SOFTWARE,[5] where the organization of a subsystem for managing transactions against financial accounts is described. Figure 2 shows how this subsystem may be modeled using the new notation for meta-level aggregation.

The class definitions for the financial management classes include the following:

```
Object
    subsystem: #FinancialManager ... !
```

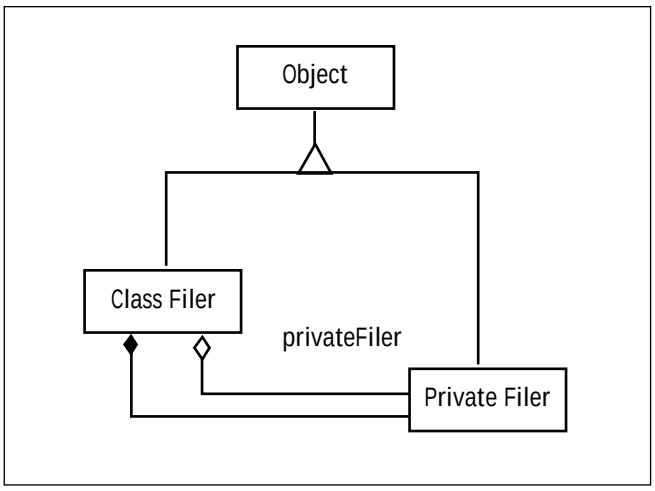

Figure 4. ClassFiler as a Façade.

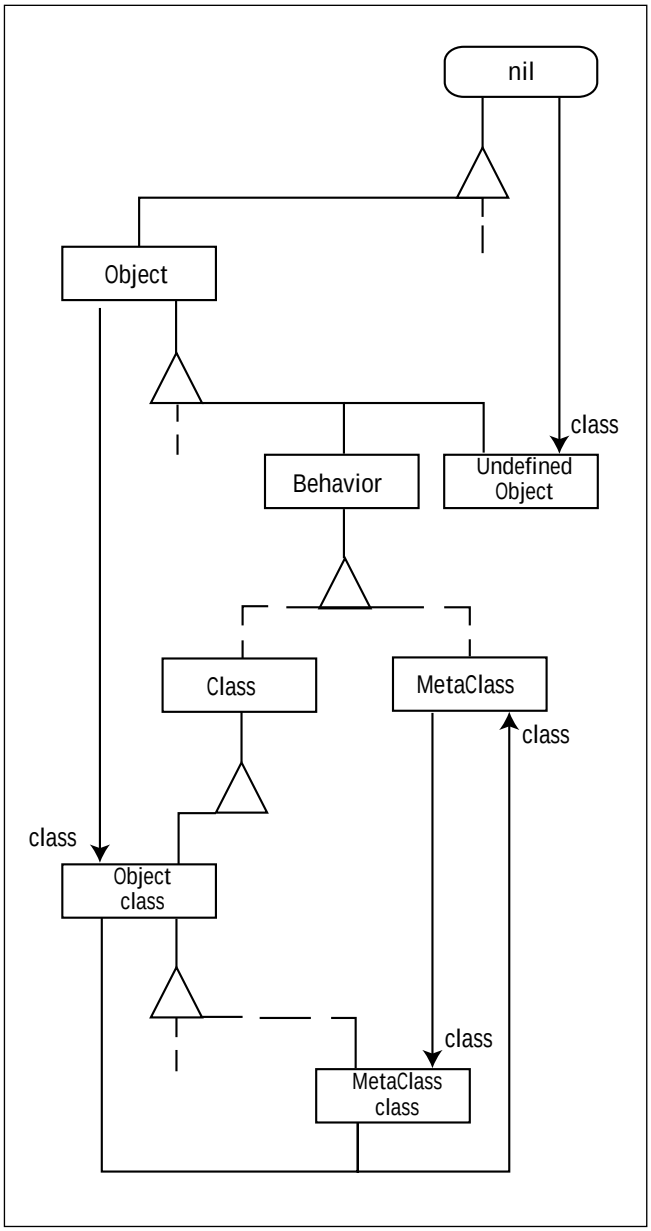

Figure 5. Baseline Behavior Classes.

```
Object
    subclass: #Account in: FinancialManager ... !

Object
    subclass: #Transaction in: FinancialManager ... !

    FinancialManager @ #Transaction
    subclass: #BalanceInquiry ... !

    FinancialManager @ #Transaction
    subclass: #FundsDeposit ... !

    FinancialManager @ #Transaction
    subclass: #FundsWithdrawal ... !

    FinancialManager @ #Transaction
    subclass: #FundsTransfer ... !
```

Subsystems and the Façade Pattern

First-class subsystems can be used to implement the Façade pattern.

The Façade pattern provides a unified interface to a set of interfaces in a subsystem. The Façade pattern defines a higher-level interface that makes the subsystem easier to use. [5]

Depending on the needs of clients, designers can either expose or hide the services provided by the private classes hidden behind the Façade.

Figure 3 shows a Façade class, which uses instances of two private classes and an instance of one public class (defined outside the subsystem). Each instance of the Façade class owns an instance of one of the private classes, while that instance owns an instance of the other private class, which in turn owns an instance of the public class.

This model also serves as an example of the visibility and scoping rules. The Façade can see classes A, B, and C. Classes A and B can see class C (which is public), but class C cannot see classes A and B (which are private).

Private Methods and Client Contracts

Smalltalk systems use classes to encapsulate the structure and state of objects. However, while Smalltalk classes encapsulate the state of their instances, they do not encapsulate their behavior. All the methods of a class are effectively public.

Traditionally, a Smalltalk developer indicates that a method is intended for the private use of the implementing class, using the notation Private at the beginning of the method comment. This convention requires the client developer to inspect the source code of the method, in order to discover whether a method is intended for public usage.

In systems that support method organization (i.e., protocols), the method developer can organize the method in a protocol whose name indicates that the methods are private. However, Smalltalk does not enforce the privacy

indicated by either of these conventions. So, client developers sometimes use the private methods anyway, and thereby create dependencies that the class designer did not intend to permit or support.

In Smalltalk, it is not always clear what such privacy means anyway. For example, should subclasses be restricted from using private methods they inherit from their superclasses? While C++ provides explicit access control mechanisms for public, protected, and private members, Smalltalk does not provide any mechanisms for access control.

It can be argued that the traditional notions of access in object-oriented systems are simplified ways of specifying the class of the clients that are permitted to use the methods of a server class. Table 1 suggests how access relates to clients.

Software developers need language facilities that provide design options. This is one of the reasons that C++ has evolved so much over the past several years.

This table formalizes a notion that has appeared repeatedly in the literature on object-oriented design: promised behavior. The classes that collaborate closely within a subsystem often exhibit promised behavior, especially when the classes in the subsystem form contractual agreements regarding their services. Thus, it would be advantageous to object system designers if object-oriented languages incorporated and enforced access mechanisms, based on client specifications to establish such formal contracts. Object-oriented languages would improve their ability to model such contracts if they were extended beyond the traditional support for only private, protected, and public access (which are supported by languages like C++ and Java). Indeed, private, protected, and public access mechanisms can be conceived of as specific kinds of promised contracts as shown in Table 2 (with respect to a given server class).

Note that in Table 2 public methods are promised to nil because the class Object and all other root classes are derived from nil—all the root classes have no superclass. Thus, public methods are available to any other defined method, whether the method is defined in a class derived from Object, or any other root class.

Access	Implied Client Specification
private	only the implementing class
protected. . . .	the implementing class and all derived classes
promised. . . .	some specific collaborating class (which need not be related by inheritance)
public. . . .	any class (without regard for inheritance)

Table 1 represents how access relates to clients.

Access	Equivalent Contract
ServerClass private	ServerClass promisedTo: ServerClass only
ServerClass protected	ServerClass promisedTo: ServerClass any
ServerClass public	ServerClass promisedTo: nil

Table 2 represents private, protected, and public mechanisms.

Private Classes for Private Methods

The following discussion describes how you can use private classes to implement private methods— even without direct language support for private methods. First, build the public interface using an instance of a subsystem class. The subsystem instance contains a single instance variable, and the instance variable contains an instance of a private class. The private class contains those methods you want hidden. When the public class (the subsystem) is instantiated, it creates and holds an instance of the private class. Each of the public methods (in the public class) uses the private methods supplied by the instance of the private class. Figure 4 shows this arrangement using an object model.

The class definitions for this Façade include the following:

```
Object
    subsystem: #ClassFiler
    instanceVariables: 'privateFiler'
    classVariables: ''
    poolDictionaries: '' !

Object
    subclass: #PrivateFiler
    in: ClassFiler
    instanceVariables: 'behavior'
    classVariables: ''
    poolDictionaries: '' !
```

Implementation

This section outlines how the facility for defining subsystems and private classes can be added to Visual Smalltalk. We will focus on those aspects of the Behavior classes that change when subsystems are added. First, note how the baseline Behavior classes are organized in Figure 5. Behavior inherits from Object. Class and MetaClass inherit from Behavior. Object class inherits from Class. The other metaclasses of the subclasses of Object inherit from Object class.

Generally speaking, the class and metaclass inheritance hierarchies parallel each other. Thus, for example:

```
Point superclass == Object
Point class superclass == Object class.
```

However, there is an anomaly at class Object, where

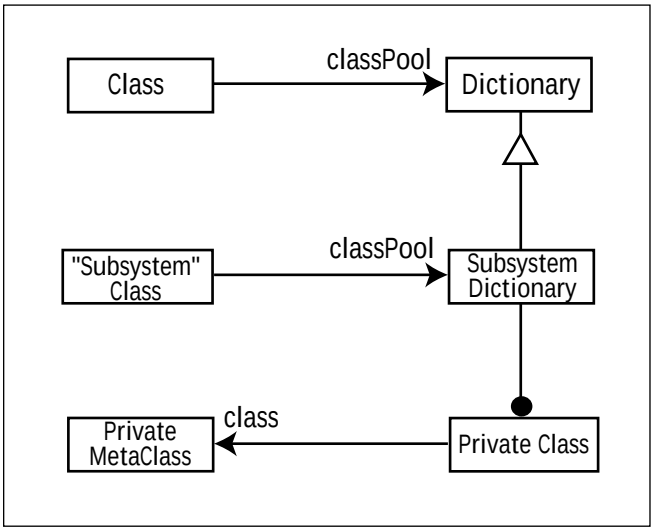

Figure 6. Class vs. "Subsystem" Class.

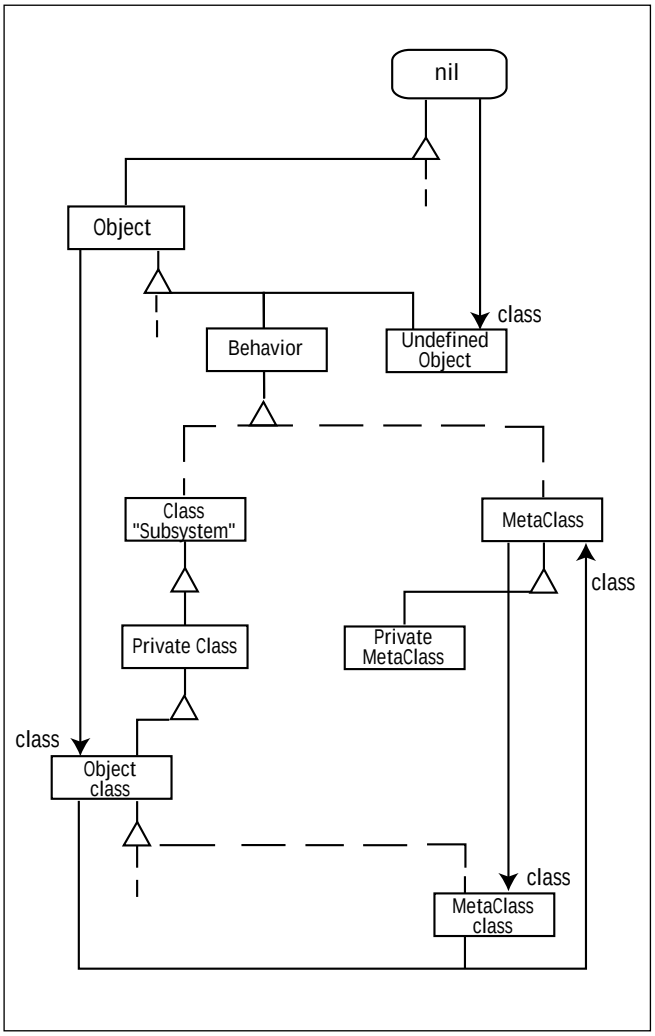

Figure 7. Behavior Extensions for Private Classes.

```
Object superclass == nil
Object class superclass == Class.
```

Subsystems and private classes require some minor alterations to these baseline relationships. As noted previously, each subsystem is a class. While an ordinary class uses a Dictionary for its **classPool,** a subsystem class uses a Subsystem Dictionary. Each Subsystem Dictionary provides a unique domain for the private classes and class variables of the subsystem. Private class names are mapped to Private Classes, while class variable names are mapped to class variables. Figure 6 provides a model of these relationships.

Each Private Class knows its class (a Private MetaClass). Each Private MetaClass knows its subsystem (a subsystem class). Thus, indirectly, each Private Class knows the enveloping subsystem class. Given the foregoing relationships, Figure 7 shows the relationships for the new Behaviors.

In particular, note how

```
Object class superclass == PrivateClass.
```

This relationship replaces the normal baseline relationship, where

```
Object class superclass == Class.
```

Because each private metaclass knows the subsystem to which it belongs, the compiler can identify the scopes that enclose the private behaviors (class and metaclass). This simplifies changes to the compiler interface to extend the visibility rules and resolve class names into classes.

CONCLUSION

Benefits of Subsystems and Private Classes

Software developers need language facilities that provide design options. This is one of the reasons that C++ has evolved so much over the past several years. Some of the recent additions (nested classes, templates, namespaces, and runtime type information) show progress toward features found in pure object-oriented systems like Smalltalk, and even some advances over features in Smalltalk.

Two of these C++ features directly address the class naming problem in complementary ways: namespaces and nested classes (i.e., private classes). This article has considered how support for private classes can be added to Smalltalk in conjunction with first-class subsystems. Subsystem classes provide an additional design dimension beyond that provided by ordinary classes. Subsystems and their private classes permit you to:

- resolve class name conflicts with separate name spaces;
- integrate separately developed class libraries;
- organize collaborations between classes into first-class subsystems;
- implement the Façade pattern; and
- define private methods only visible to those in a public interface (subsystem) class. **S**

References

1. Wirfs-Brock, A., Wilkerson, B., "An Overview of Modular Smalltalk," OOPSLA Conference Proceedings, ACM, September 1988.

2. Beaton, W., "Name Space in Smalltalk/V for Win32," *The Smalltalk Report* 4(1), SIGS Publications, New York, NY, September 1994.
3. Boyd, N., "Modules: Encapsulating Behavior in Smalltalk," *The Smalltalk Report* 2(5), SIGS Publications, New York, NY, February 1993.
4. Wirfs-Brock, R., Wilkerson, B.,Weiner, L., DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice-Hall, Englewood Cliffs, NJ, 1990.
5. Gamma, E., Helms, R., Johnson, R., Vlissides, J., DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED ARCHITECTURE, Addison-Wesley, Reading, MA, 1995.

TRADEMARKS

Nik Boyd has been developing object systems since 1987, when he founded 3rd Person Software (formerly known as XoteryX). In 1993, he released Package Manager/V through The Smalltalk Store. During 1996, he released Package Librarian/V. His experience with OOP includes work with several ParcPlace-Digitalk Smalltalk versions and platforms, as well as work with C++. His research interests focus on tools and techniques that support object-oriented software engineering. Nik may be contacted at 74170.2171@CompuServe.com.