The Smalltalk Report

The International Newsletter for Smalltalk Programmers

February 1993 **Volume 2 Number 5**

MODULES: ENCAPSULATING BEHAVIOR IN SMALLTALK

By Nik Boyd

Contents:

This article proposes a new view of modules and how they may be added to the Smalltalk programming system. Modules provide a way to control the visibility of shared names. Modules also provide a way to hide the detailed collaborations among a group of Smalltalk classes organized as a subsystem. The organizing principles of classes and modules are orthogonal. Thus, modules also can be used to safely extend existing baseline classes.

The concept of a module and modular software development has existed for many years. A variety of programming systems has provided support for using separate name spaces to control the visibility of names used in a program. Examples include Modula-2[1] and Ada.[2]

Smalltalk systems use classes to encapsulate the structure and state of objects. Because Smalltalk classes can hide their internal state and serve as centers around which program behavior may be organized, they also may be considered modular. But while Smalltalk classes can encapsulate the state of their instances, they do not encapsulate their instances' behavior.

By convention, some messages are designated as "private" for the private use of the class and its instances. However, the Smalltalk system does not enforce designated message privacy and it is not always clear what such privacy means. For example, should subclasses be restricted from using private messages they inherit from their superclasses?

Because classes are globals in the Smalltalk system dictionary, they are all visible to all other classes. This visibility is excessive and it can contribute to information overload for novice Smalltalk programmers. It also can cause class naming conflicts when a team of developers integrate their separately developed components.

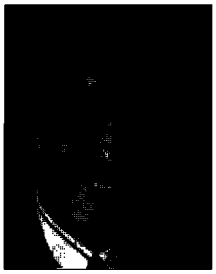
This article attempts to deal with these issues in a relatively nonintrusive manner that does not sacrifice any of the flexibility and power offered by existing Smalltalk systems.

MODULES

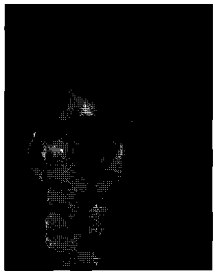
In their work on Modular Smalltalk,[3] Allen Wirfs-Brock and Brian Wilkerson describe the essential features of modules:

> Modules are program units that manage the visibility and accessibility of names. . . .
> A module typically groups a set of class definitions and objects to implement some service or abstraction. A module will frequently be the unit of division of responsibility within a programming team. . . .
> A module provides an independent naming environment that is separate from other modules within the program. . . .
> Modules support team engineering by providing isolated name spaces. . .

continued on page 4...



John Pugh



Paul White

EDITORS' CORNER

Most of you are probably satisfied with Smalltalk as a development tool. In fact, many of us feel even a bit arrogant about promoting Smalltalk as the "best" tool for developing software systems. Nevertheless, most would have to agree that we still lack an integrated process, and tools to match that process, for the entire software development lifecycle. Whether we're using CRC, Booch's notation, OMT, or something else, there still exists a "leap" from the design process to the construction of the software. Many of the tools on the market today offer little in the way of matching designs with the corresponding code.

Even more important, though, is the fact that we still don't have proper tools to allow us to go back and update the design to reflect changes in the construction. If we are to reap the benefits of the new "object-oriented lifecycle" many of us are advocating, where the design and development phases can be better integrated, we're going to need such tools. As Sam Adams and Steve Burbeck pointed out in the November/December 1992 issue of Object Magazine, "design is a continual process of discovering, evaluating, and deciding between alternatives." This can only be achieved if the costs of doing so are manageable.

One issue that has been addressed over the months in this newsletter is how best to manage visibility of objects within Smalltalk. On large development projects where teams work on subsystems to be integrated, managing the name space always proves to be a difficult task. In our feature article this month, Nik Boyd takes a new look at using modules as a vehicle for managing class library name spaces. He states that modules can be used to hide the details of the implementation of a software component consisting of a number of cooperating classes and he discusses means for implementing them.

In her column this month, Rebecca Wirfs-Brock calls for software development teams to characterize their objects. She states that such characterizations will help ensure that all team members are "in sync" and working toward a common system architecture. Through her experience, she proposes a number of terms for characterizing objects that could be adopted by your team.

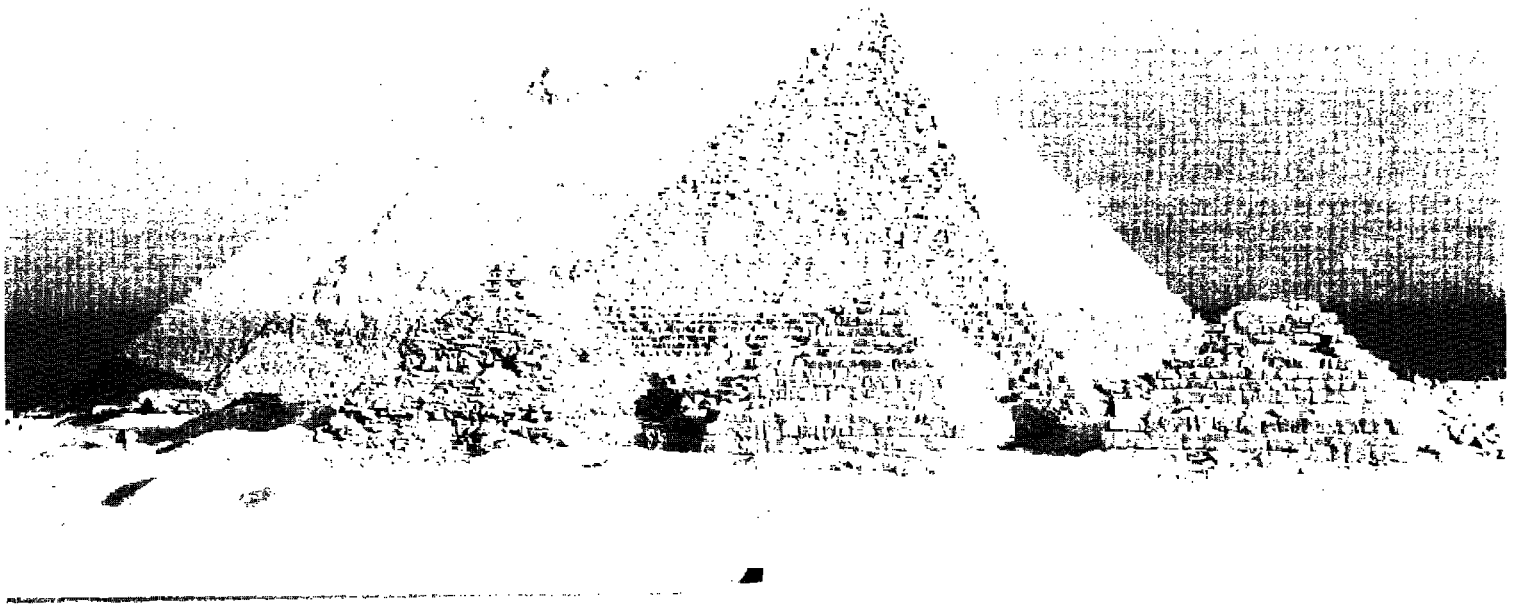
Kent Beck offers two columns in one this month. As an aside, he describes a short idiom for testing nil values in an expression. The main column calls for software developers to "describe the intent behind a piece of code" for those who will later need to understand it to reuse it. Kent suggests that what is needed is a "pattern language" capable of describing these intentions at a variety of levels.

Juanita Ewing's "Getting Real" column addresses the problem common to all computing languages—how best to deal with defining and using both constants and default values. As she points out, to develop a code that will be reusable, default values must be defined in a consistent fashion and a mechanism must be provided for overriding them. In the "GUI" column this month, Greg Hendley and Eric Smith comment on the similarities and differences between Cooper and Peters' WindowBuilder and ParcPlace's new VisualWorks. The issues involved in copying Smalltalk objects arise regularly on USENET, and Alan Knight tackles some of these issues in his regular "comp.lang.smalltalk" column this month.

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, every month except for the Mar/Apr, July/Aug, and Nov/Dec combined issues. Published by SIGS Publications Group, 588 Broadway, New York, NY 10012 (212)274-0640.

Mailed First Class. Subscription rates 1 year, (9 issues) domestic, $65, Foreign and Canada, $90. Single copy price, $8.00. POSTMASTER: Send address changes and subscription orders to: The Smalltalk Report, Subscriber Services, Dept. SML, P.O. Box 3000, Denville, NJ 07834. Submit articles to the Editors at 91 Second Avenue, Ottawa, Ontario K1S 2H4, Canada.

Like ENVY/*Developer*, Some Architectures Are Built To Stand The Test Of Time.



ENVY/*Developer*: The Proven Standard For Smalltalk Development

An Architecture You Can Build On

ENVY/*Developer* is a multi-user environment designed for serious Smalltalk development. From team programming to corporate reuse strategies, ENVY/*Developer* provides a flexible framework that can grow with you to meet the needs of tomorrow. Here are some of the features that have made ENVY/*Developer* the industry's standard Smalltalk development environment:

Allows Concurrent Developers

Multiple developers access a shared repository to concurrently develop applications. Changes and enhancements are immediately available to all members of the development team. This enables constant unit and system integration and test – removing the requirement for costly error-prone load builds.

Enables Corporate Software Reuse

ENVY/*Developer*'s object-oriented architecture actually encourages code reuse. Using this framework, the developer creates new applications by assembling existing components or by creating new components. This process can reduce development costs and time, while increasing application reliability.

Offers A Complete Version Control And Configuration Management System

ENVY/*Developer* allows an individual to version and release as much or as little of a project as required. This automatically creates a project management chain that simplifies tracking and maintaining projects. In addition, these tools also make ENVY/*Developer* ideal for multi-stream development.

Provides 'Real' Multi-Platform Development

With ENVY/*Developer*, platform-specific code can be isolated from the generic application code. As a result, application development can parallel platform-specific development, without wasted effort or code replication.

Supports Different Smalltalk Vendors

ENVY/*Developer* supports both Objectworks\Smalltalk and Smalltalk/V. And that means you can enjoy the benefits of ENVY/*Developer* regardless of the Smalltalk you choose.

For the last 3 years, Fortune 500 customers have been using ENVY/*Developer* to deliver Smalltalk applications. For more information, call either Object Technology International or our U.S. distributor, Knowledge Systems Corporation today!

Object Technology International Inc
2670 Queensview Drive
Ottawa, Ontario K2B 8K1

Ottawa Office
Phone: (613) 820-1200
Fax: (613) 820-1202
E-mail: info@oti.on.ca

Phoenix Office
Phone: (602) 222-9519
Fax: (602) 222-8503

Knowledge Systems Corporation
114 MacKenan Drive, Suite 100
Cary, North Carolina 27511
Phone: (919) 481-4000
Fax: (919) 460-9044

While providing many potential improvements to Smalltalk, the Modular Smalltalk system does not implement modules as first-class objects. Like many other programming systems, the Modular Smalltalk system uses modules only for organizational purposes. This article proposes a different view of modules as a special kind of Smalltalk class.

MODULES FOR SMALLTALK

The definition of a normal Smalltalk class includes a reference to a superclass, the name of the new subclass, and the names of any new instance and class variables added by the new subclass. Class variables are shared by all the instances of a class and are visible to all its methods and subclasses, if any.

In addition, the new subclass can provide its methods with access to named objects that are shared on a subscription basis. Certain names in the Smalltalk system dictionary are bound to global pool dictionaries that contain these sharable named objects. The new subclass can subscribe to these objects by including selected global names in its list of pool dictionaries. For example, a File class might be defined using the following message:

```
Object subclass: #File
    instanceVariableNames:
        'directory fileId name '
    classVariableNames:
        'PageSize '
    poolDictionaries:
        'CharacterConstants '!
```

Modules may be added to Smalltalk in a relatively straightforward manner. Details of how this can be done are presented in a later section. For now, we can say that each module is a class containing a name space, called its domain, instead of simply a pool of class variables.

There are several new messages for defining modules and the private classes contained in their domains. The definition of a module for managing an inventory might use the following message:

```
Object moduleSubclass: #InventoryManager
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

A new private class can be added to the domain of the InventoryManager class using the message:

```
Object subclass: #InventoryItem
    in: InventoryManager
    instanceVariableNames:
        'partNumber partName quantity '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

In order to add a new private subclass of InventoryItem, we send the name of the private class (#InventoryItem) as a message to the InventoryManager module:

```
InventoryManager
InventoryItem subclass: #FloorItem
    instanceVariableNames:
        'storeLocation '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

The issues involved in this breaking of the module encapsulation will be considered further in a later section.

Modules can be used to create nested subsystems. The following message creates a nested module for managing accounts in the InventoryManager module class:

```
Object moduleSubclass: #AccountManager
    in: InventoryManager
    instanceVariableNames: ' '
    classVariableNames: ' '
    poolDictionaries: ' '!
```

Figure 1 depicts the structural relationships between classes in the InventoryManager module. Note that the graphic design notation of OMT[4] has been extended slightly to show what classes are encapsulated inside a module class. The rounded rectangles represent module domains. Note that the Smalltalk system dictionary also is considered to be the system domain.

ENCAPSULATING PRIVATE BEHAVIOR

Modules provide three ways of encapsulating private behavior, all of which are based on their ability to encapsulate private classes:

- class groups (systems)
- baseline class extensions
- private methods

Each of these options will be discussed in the following sections.

PACKAGING OBJECT SYSTEM DESIGNS

One advantage of modules is that they provide a way for developers to package systems of components. During the design of a system of objects, groups of classes often know of each other

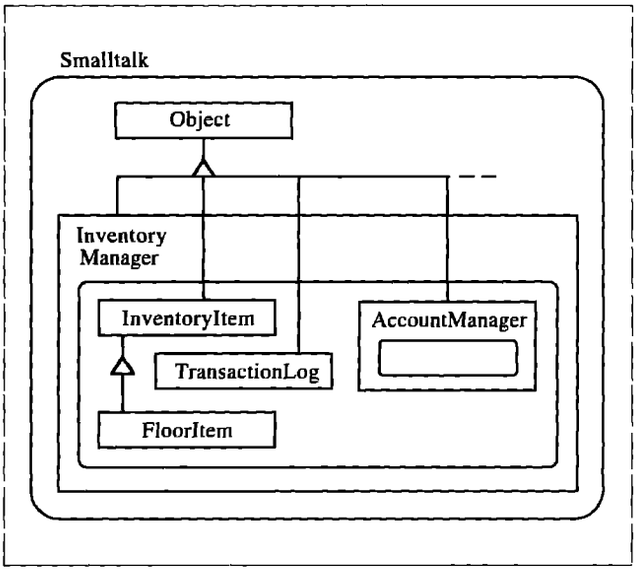


Figure 1. Structural relationships between classes.

explicitly and collaborate closely to produce some complex behavior. Such subsystems are described informally in DESIGNING OBJECT-ORIENTED SOFTWARE[5]:

> **Subsystems are groups of classes, or groups of classes and other subsystems, that collaborate among themselves to support a set of contracts. From outside the subsystem, the group of classes can be viewed as working closely together to provide a clearly delimited unit of functionality. From inside, subsystems reveal themselves to have complex structure. They consist of classes and subsystems that collaborate with each other to support distinct contracts that contribute to the overall behavior of the system....**
> **Subsystems are identified by finding a group of classes, each of which fulfills different responsibilities, such that each collaborates closely with other classes in the group in order to cumulatively fulfill a greater responsibility....**
> **There is no conceptual difference between the responsibilities of a class, a subsystem of classes, and even an entire application; it is simply a matter of scale, and the amount of richness and detail in your model...**

This article goes beyond the conceptual to assert that there is no practical difference between the responsibilities of a class and a subsystem of classes when the subsystem is implemented as a module. The module class acts as a capsule around the subsystem of classes enclosed within the module domain.

Such packaging supports some of the practices of good software engineering. Implementation details can be localized, encapsulated, and scoped. Just as good object designs organize state and behavior into classes, systems of objects that are closely coupled, or that cooperate to provide some overall set of services, can be organized into modules.

EXAMPLE SYSTEMS

DESIGNING OBJECT-ORIENTED SOFTWARE gives several examples of object system design based on responsibilities, two of which are described in this article with just their class definitions. The first example already has been presented. The InventoryManager depicted in Figure 1 was derived from the Inventory subsystem described on pages 146–148 of the above book. Pages 151–152 describe the organization of a subsystem for managing transactions against financial accounts. Figure 2 shows how this subsystem might be organized as a module. The classes for this system could be defined using the following messages:

```
Object moduleSubclass: #FinancialManager
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''!
Object subclass: #Account
    in: FinancialManager
    instanceVariableNames:
        'accountID balance'
    classVariableNames: ''
    poolDictionaries: ''!
Object subclass: #Transaction
    in: FinancialManager
    instanceVariableNames:
        'account'
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #BalanceInquiry
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #FundsDeposit
    instanceVariableNames:
        'amount'
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #FundsWithdrawal
    instanceVariableNames:
        'amount'
    classVariableNames: ''
    poolDictionaries: ''!
FinancialManager
Transaction subclass: #FundsTransfer
    instanceVariableNames:
        'amount targetAccount'
    classVariableNames: ''
    poolDictionaries: ''!
```

EXTENDING BASELINE SMALLTALK CLASSES

Modules provide a safe way to extend and package changes to baseline classes in the Smalltalk system domain. Figure 3 shows how a private version of the String class can transparently subclass its baseline version so as to extend it.

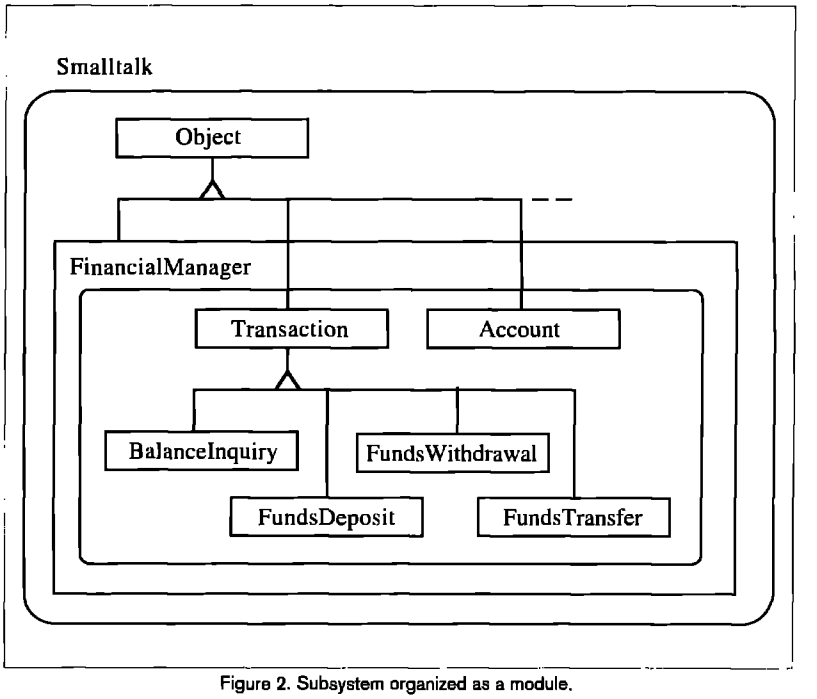


Figure 2. Subsystem organized as a module.

ModuleA is a moduleSubclass of class Object and SubclassB is a private class inside the domain of ModuleA. The private String class inside the domain of ModuleA is a private subclass of the baseline String class:

```
Object moduleSubclass: #ModuleA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''!
    Object subclass: #SubclassB
        in: ModuleA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''!
    String variable Byte Subclass: #String
        in: ModuleA
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''!
```

The private String class extensions are visible to methods in both ModuleA and SubclassB but not to classes outside of ModuleA in the Smalltalk system domain, such as SubclassC.

One drawback exists in the above example. The compiler creates constants for literals using the baseline classes: SmallInteger, Float, String, Symbol, and Array. Unlike Objectworks\Smalltalk, Smalltalk/V presently does not include the source code for its compiler. Because the Smalltalk/V compiler has not been extended to use the privatized versions of baseline classes it uses for literals, the private String class needs to create instances by copying baseline strings. For example, if we want SubclassB to use a private String for some operation, it will need to create it using:

```
"private" String copyFrom:
    'a constant string'
```

The compiler creates a constant string that is an instance of the baseline String class. The private String class creates an instance of itself that is a copy of this string constant. Given an instance of the private String class, the extended private string operations may be performed on it.

Given access to the source for the compiler, this small defect could be rectified. Then all the baseline classes, including those that the compiler uses for literals, could be extended transparently by private subclasses.

ENCAPSULATING PRIVATE METHODS

Modules can be used to hide the private methods of a class. To do this, a pair of classes is used to divide the public methods from the private ones. The public class is a module whose methods provide its public interface. The private methods are hidden in a private class inside the module domain. The private class can have the same name as the module class.

Figure 4 depicts an example of how this principle can be applied. Because of its simplicity, the full code for this example can be found in Listing 1. The module class ClassFiler is derived from the standard Smalltalk class ClassReader. This class is used to file Smalltalk source code in and out of the system, usually using an instance of class FileStream.

The ClassFiler module class has a single instance variable: privateSelf. When an instance of the module class is created, privateSelf is set to reference an instance of the private ClassFiler class. All the public methods of the module delegate private messages to privateSelf. Instances of the module class serve as proxies that hide the private behavior of the module class.

To maintain encapsulation, public methods in the module class can check the answers that come back from privateSelf. Any answer that is identical to privateSelf should be answered as self (the module instance) instead.

This technique provides true encapsulation of private methods of the class with a small amount of overhead in time (the delegation and answer checking) and space (the extra instance privateSelf).

continued on page 19

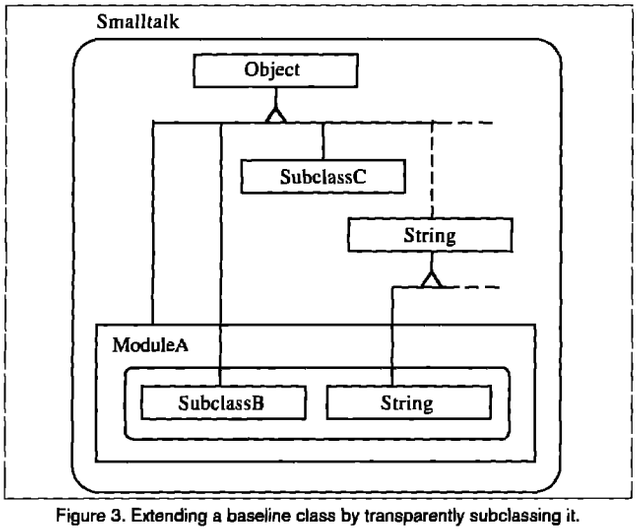


Figure 3. Extending a baseline class by transparently subclassing it.

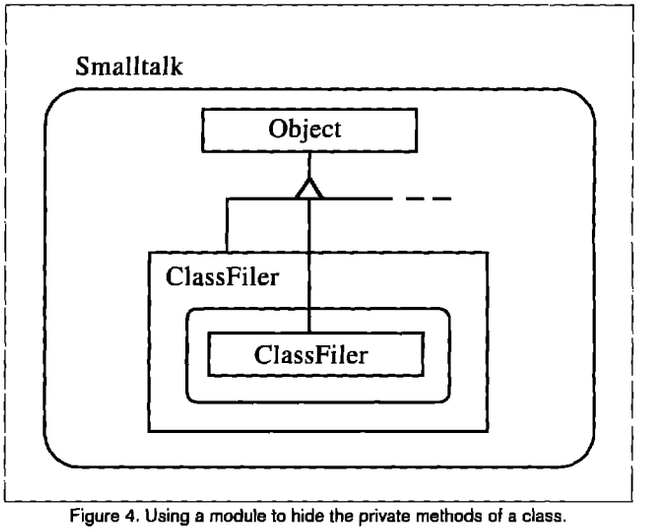


Figure 4. Using a module to hide the private methods of a class.

Rebecca Wirfs-Brock

Characterizing your objects

In this column I'll describe some vocabulary I find useful to characterize objects. Building an application involves teamwork and cooperation. Melding classes designed by individuals into a consistent system of cooperating objects requires that team members work toward a common system architecture. Team members need to share an understanding of what constitutes well-designed classes and subsystems, and what are acceptable patterns of object interactions.

Choices between perfectly acceptable alternatives must be made consistently across classes designed by different people. Achieving a consistent pattern of object communication first requires team members to use a common vocabulary for describing objects and their communication patterns. Once team members are talking the same language, they can have meaningful discussions about desirable interaction styles. Decisions then can be made based on sound engineering practices that meet business requirements.

STEREOTYPING OBJECT ROLES

Objects in our design can be either involved, active participants in many conversations, or by design play a more docile role, responding only when asked and taking a supporting role. Between these two extremes are many shades of behavior. I find it useful to classify objects according to their primary purpose as well as their modus operandi.

Here are two ways to characterize object roles:

- **Business Objects.** Objects whose primary purpose is to model necessary aspects of a concept that would be familiar to a user of the software we design. If we were designing an Automated Teller Machine for a bank, we might have Bank Customer, Bank Account, and Financial Transaction objects. If we were designing an oscilloscope we might model Triggers, Waveforms, or Timebases. These types of objects are also commonly referred to as domain objects because they correlate directly with concepts in the users' domain.
- **Utility Objects.** These are generally useful, non-application-specific objects. Smalltalk programming environments come with many generically useful classes. Classes for structuring other objects, such as Set, Array, Dictionary, and classes representing numbers or strings fall into this category.

There are compelling reasons for application developers to create additional utility objects. For several projects I've worked on, specific individuals were assigned direct responsibility for creating, publishing, and ensuring that utility objects were appropriate to the task and properly used. It is possible to create and effectively incorporate utility objects into the application throughout development and software construction.

It is extremely useful to design new utility objects that explicitly support system policies or common application programming practices. For example, we have created classes that stylize error handling and sequencing of processing steps; classes that model ranges of set table values, increments, and units of measurement; and classes that monitor detectable external conditions. Once designed, these objects can be used in many places within an application.

STEREOTYPING OBJECT BEHAVIORS

A number of researchers and design methodologists have coined terms for describing objects according to the way they operate. My list of useful terms isn't merely a composite of all common terms in the current literature. I continue to make finer distinctions after reflecting on past experiences and tackling new design projects. Periodic updating is needed to reflect new ways of constructing software that accomplishes new tasks.

Following are useful ways to classify object behavior.

Controlling objects

Controlling objects are responsible for controlling a cycle of action. This cycle can be either repetitive, with conditional branching logic, or initiated and executed once on detection of a certain set of events or circumstances. Controlling objects can initiate and control ongoing systemwide activity or iterate over a minor application task.

The original Smalltalk-80 user interface presented a stylized three-way collaboration between Model, View, and Controller objects. Controller objects were responsible for responding to user directives, such as mouse clicks or keystrokes, and initiating appropriate responses. Views displayed the current state of the application and model objects were application-specific objects.

I use a broader definition than that implied by Smalltalk-80 Controller objects. Controlling objects need not be spurred to action only on behalf of user directives. Controlling objects can be found and created for many parts of an application where a

cycle of activity is initiated, sequenced, and, sometime later, possibly completed.

For example, in the design of an Automated Teller Machine, an ATM object can have responsibility for initializing and sequencing system interactions with a bank customer. A further design refinement can add the concept of a SessionController object, which controls the sequence of activities by a single bank customer wishing to carry out one or more transactions with the bank. At a lower level, there may be network controller objects responsible for handling network traffic between the application and the communication network.

Coordinating objects

Coordinating objects are the traffic cops and managers within a system. Coordinators often pair client requests with desired services (or, rather, objects performing a requested service). In my early object design experience, I would append Manager to the names of these objects. FontManager and StyleManager are two example class names. I used to feel uncomfortable creating objects whose primary behavior was being idle until someone needed something, then helping to establish the connection between two other objects that would collaborate to actually perform some useful function. I now realize that these coordinators proved their worth simply by eliminating the need to hard-wire direct references between objects.

In another common design pattern, a coordinating object may respond to a request by briefly establishing an appropriate context, then delegating a request to one or more objects within its sphere of influence. For example, in the ATM design, the Session Manager first would determine which transaction the bank customer wished to perform, then create the appropriate transaction object for delegating the responsibility to gather additional information from the bank customer (such as amount to withdraw if it were a Withdraw Transaction), and then perform the transaction.

A coordinating object also may control a sequence of actions. It is often logical to blend coordinating and controlling functions in the same objects. A reasonable design for the Session Manager object is to give it the responsibility for creating and handling a series of bank customer transactions. A bank customer typically can perform transactions until indicating a desire to terminate the session, causing our application to print a receipt of all transactions and return the customer's card.

Structuring objects

Objects with structuring duties primarily maintain the relationships between application objects. In many applications, business objects have very complex structural relationships. Let's take a simplistic real-world example of a file cabinet containing folders that hold documents. A file cabinet simply holds folders that may be tabbed and labeled, and the folders merely contain their contents. The documents themselves are of interest.

In an object design, I add more or less behavior to objects to meet business requirements and to suit my personal tastes. I can design File Cabinets to do more than organize their contents. A File Cabinet could know when any folder was last referenced, or how much room is left in the cabinet. When I classify an object as primarily a structuring object, I think first and foremost about what relationships it should maintain between other objects and how it should do so, and secondarily what (if any) additional behavior might be appropriate and useful for it to have.

Informational objects

Sometimes objects are created to hold values that can be requested by many different kinds of application objects. I don't want to get into an in-depth discussion of design and programming techniques to eliminate globals or minimize dependencies on hard-wired values in code. However, at times it can be useful to create objects that are responsible for yielding information. In procedural programming languages, we have the ability to declare constant values. In object designs, informational objects are an equivalent concept.

Service objects

A service object typically is designed to perform a single operation or activity on demand. A well-designed service object provides a simple interface to a clearly defined operation; it should be easy to set up and use. Pure service objects often are the products of a highly factored design. Such a design consists of many classes of objects having highly specialized behaviors.

One reason to create service objects is to facilitate optional or configurable software features. The argument for this design strategy goes something like this: It is easier to configure a product's features by adding or removing entire classes of objects than it is to add or remove class behaviors.

As more behavior is added to a class, it can become complex to integrate new features with existing code. Optional functionality needs to be implemented in a way that guarantees pre-existing code doesn't break. Test suites and internal consistency checks become important.

When services are placed in specialized service classes, the design task shifts to creating an appropriate role and interface to the service object, which must balance the client's control over the service's performance with simplicity and ease of use.

An operation may be so complex to perform that it warrants creating many objects. A single object can be designed to provide the public interface to this service, hiding most of the details from the rest of the application.

Useful services can be packaged into distinct objects. These service objects might be designed so as to be useful in a variety of contexts, perhaps by being easy to extend or customize. We could design our ATM transaction objects to know precisely how to print information about the transaction on a receipt. Alternatively, we could design a Report object that provides printing and formatting services for the transaction object.

Interface objects

Interface objects are found at the boundaries of an object-oriented application. They can be designed to support communi-

cations with users, other programs, or externally available services. Interface objects come in many sizes, shapes, and flavors, and at many conceptual levels.

Interface objects can be designed to support an ongoing two-way communication between some external entity. For example, in the ATM application we have a number of physical devices such as Receipt Printer, Cash Dispenser, and Card Reader. In our design, all these devices would have interface objects that define a high-level interface to the services they provide. A Cash Dispenser object might define as message to dispense cash, return the cash balance, or adjust the balance (as a result of dispensing cash or adding more money to the machine).

Interface objects can be designed to translate external events or requests into messages fielded by interested application objects. For example, many external events need to be handled by the ATM system. To name a few: jamming of cash in the Cash Dispenser, failure of the door to close, the Receipt Printer running out of paper, etc. The list isn't endless, although responsible objects (the most likely candidates are appropriate interface objects) need to field those events and respond appropriately.

Or they can be designed to provide a narrow interface. For example, a menu presents a number of options and returns a user's preference. User interface objects typically support a highly stylized dialogue between the user and the system.

Interface objects are responsible for bridging the non-object world and the object world of messages and objects. When I think about interface object design, I focus first on those objects considered by the remaining applications to define the interface to the outside world. I realize that a great many details can and should be encapsulated by these interface objects. The key is to hide these details and provide a sufficiently abstract interface.

MOVING OBJECT DESIGNS ALONG THE BEHAVIORAL CONTINUUM

Given that we have a sufficiently rich vocabulary for describing object roles and behavioral patterns, we need to establish a context for applying these terms. Once we have done so, we need to evaluate our emerging design and select among alternatives. First it is useful to distinguish at what conceptual design level an object should belong (as opposed to where it is currently placed). Is it a high-level object or does it provide low-level services? Does it a play a significant or relatively insignificant role?

Once we determine this conceptual level, we can easily characterize an object's role as business or utility. Examining behaviors and building cleanly defined objects takes more time. Objects don't always fall into a single behavioral category, nor do I expect them to. For instance, objects often blend behaviors of controlling and coordinating. Another common pattern is to blend behaviors for structuring and providing services into the same object.

I do find it useful to ask whether an object is assuming too much responsibility, and whether it would be more appropriate to create new classes of objects to share the load. I also note whether a design choice causes an object's behavior to shift one way or the other on a behavioral continuum. Has an object become too active or passive? Is it perhaps taking on too many behaviors by assuming both a coordinating role as well as performing a useful service? Would it simplify the design to subdivide an object's responsibilities into smaller, simpler concepts? What would be an appropriate pattern of collaboration between that object and newly defined service objects?

When I look at rebalancing behaviors, I tend to consider the current behavior definitions for a group of collaborating objects belonging to roughly the same conceptual level. My goal is to understand and develop an appropriate distribution of control logic and responsibility among colloborators. Design creativity and individual preferences needn't be sacrificed during this assessment process. However, readjusting object behaviors needs to be purposefully done. In my next column I will discuss some object interaction styles as well as strategies and reasons for choosing between them. ■

Rebecca Wirfs-Brock is Director of Object Technology Services at Digitalk and co-author of DESIGNING OBJECT-ORIENTED SOFTWARE. *She has 17 years' experience designing, implementing, and managing software products, with the last eight years focused on object-oriented software. She managed the development of Tektronix Color Smalltalk and has been immersed in developing, teaching, and lecturing on object-oriented software. Comments, further insights or wild speculations are greatly appreciated by the author. She can be reached via email at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 7585 S.W. Mohawk, Tualatin, OR, 97062.*

THE BEST OF comp.lang.smalltalk

Alan Knight

Copying

Copying objects ought to be easy. After all, objects are just bits in the machine and those are easy enough to copy. Besides, objects are encapsulated, so copying shouldn't have to worry about anything outside the current object. Unfortunately, it's not always that simple. Complications can arise from details of Smalltalk's implementation and the object structure and from interactions with inheritance.

OBJECT IDENTITY

In Smalltalk, each object has a unique identity independent of the value it represents. In other words, Smalltalk variables don't hold objects but references to objects. Several different variables can refer to the same object; if a change is made to that object, the changed value is visible through all those variables.

This is also known as "aliasing" because the same object can have several different names, or "reference semantics" because the variables refer to the objects. This is in contrast to "copying semantics" where each variable has (or at least appears to have) its own copy of the object.

In pure functional languages, aliasing is eliminated. The values of instance variables in existing objects cannot be changed and new objects with different values must be created instead. Functional programmers would say that this is a good thing because it eliminates many confusing errors associated with aliasing. Non-functional programmers might say that removing aliasing entirely also eliminates many useful programming techniques but few would deny that copying semantics can be useful sometimes.

Some Smalltalk classes have copying semantics, including numbers, characters, booleans, and symbols. Operations on these types of objects do not modify the internal values of the instance but create a new instance as their result. Even though numbers can be aliased (as almost all Smalltalk objects can), there are no operations that can change the internal state and reveal the aliasing. The need to allocate new numbers for each operation results in poorer performance for numerically intensive applications but makes the behavior of numbers much more simple and predictable.

Complications

The previous section contains a number of half-truths. It's not really true that no operations modify classes with copying semantics. Meta-operations like become: and instVarAt: can get around these restrictions and it's possible to add methods that modify the internal state of some of these classes. In addition to seriously messing up your image, these facilities can expose significant differences in the behavior of these classes.

The most important difference, for copying purposes, is between SmallIntegers and all other objects. SmallIntegers are the most primitive entities in Smalltalk and really do have copying semantics, which the other classes just pretend to have.

The trick is that Smalltalk variables actually hold a 32-bit quantity, one bit of which is a flag. If the flag is set, the object referred to is a SmallInteger and the remaining 31 bits are its value. If the flag is not set, then it is some other kind of object and the remaining 31 bits are the machine address of that object.

If you copy the bits stored in a variable holding a SmallInteger, you actually get a copy of the SmallInteger. If the variable holds an object, then you get a copy of a reference to the object. This is the kind of implementation detail that you normally shouldn't have to think about, but it does explain a number of otherwise confusing things. For example, if you've ever wondered why become: doesn't work on SmallIntegers but does work on LargeIntegers, or why:

```
10 == 10
```

evaluates to true, but:

```
10 factorial == 10 factorial
```

comes out false, here is the explanation:

Become: can't work on SmallIntegers because it works by changing object references. SmallIntegers don't have object references, so there's nothing to be interchanged. In fact, since the parameter passing mechanism in Smalltalk is to copy these 32-bit fields described above, the become: operation doesn't even get the original SmallIntegers to change but only a copy of their values on the stack.

The == operation compares these same 32-bit quantities for equality. For SmallInteger 10, the bit patterns are exactly the same, so == is true. 10 factorial is a LargePositiveInteger, and since both sides of the expression are evaluated separately, we get two separate instances of LargePositiveInteger, which are equal (=) but not identical (==).

Shallow copy

How does this affect copying? The default copy implementation in Smalltalk is the "shallow copy," which just creates a new instance with exactly the same bits as in the old instance. This means we get a genuine copy of SmallIntegers and a shared reference to all other objects. Sometimes this is what you want but it also can be very confusing. For example, Richard Bentley (dik@comp.lancs.ac.uk) poses the frequently asked question:

> **Could somebody please explain to me how copy is supposed to work. To me, if I take a copy of (say) a Dictionary, the copy should not just have pointers to the original Dictionary's instance variables, so that if I change a value in my copy, the original is also changed.**
> **Is this how copy is supposed to work? If I want a deepCopy of a composite object (one that references other objects using instance variables), how should I go about it ?**

Deep copy

In many cases, a deep copy is more intuitive than the one-level shallow copy. Deep copying has its own complications, though, and it's not possible to provide a single implementation that makes sense for all classes.

Digitalk provides an implementation of deepCopy that makes a copy of an object with shallow copies of all its instance variables. This is deeper than shallow copy but it just pushes the problem down one level. This wouldn't work properly in the dictionary example either because the instance variables of a dictionary are not the keys and values but the associations that hold them. They also provide an implementation of deepCopy specific to Dictionary, which does "the right thing." Such special implementations are required for quite a few classes, and still leave open questions like "How do I copy a dictionary of dictionaries?"

ParcPlace used to provide a recursive implementation of deepCopy, which would copy an object and make deep copies of all its instance variables, recursing until it reached primitive objects. This also has problems, as Bruce Samuelson (bruce@ling.uta.edu) points out:

> **ParcPlace has been phasing out support for deepCopy because of theoretical problems such as infinite recursion for circular structures.**

Jan Steinman (steinman@hasler.ascom.ch) adds:

> **That's not good enough! #deepCopy has practical problems, such as chewing up memory when you least expect it. (Try to deepCopy a SortedCollection, for instance, which holds a BlockClosure, which holds a CompiledLocalBlock, which holds a metaclass, which links in the entire class tree. . . .)**
> **There are numerous solutions for avoiding infinite recursion, the simplest of which (context query) does not even require any additional state.**
> **I find #deepCopy so useful that I've implemented #deepSize, a "better BOSS," and lots of other deep things. They can be slow memory hogs, but if you use such things within their practical limitations, what's the problem? When #deepCopy goes away, I'll put it back!**

The phrase "context query" hides a very clever trick that takes advantage of Smalltalk's reflective capabilities to avoid infinite recursion. Using the thisContext pseudo-variable in ParcPlace Smalltalk, it is possible to examine the stack of the currently executing process. This information can be used to determine whether an object already has been visited (and abort the recursion if it has). Jan Steinman has promised to write an article for THE SMALLTALK REPORT describing these tricks in detail. Similar tricks should be possible in Digitalk implementations but the interface to the process stack is not as well-documented, so it would take a bit more investigation.

Do it yourself

In general, if you want a copy routine that does "the right thing" for a particular class, you have little choice but to write it yourself. There isn't a universal definition of what the right thing is, and it may even vary for the same class from application to application. The problem of copying complex objects with circular references (e.g. a Graph) is equivalent to the problem of storing and retrieving an object from disk. In fact, if I have objects that can be written to a file, it's sometimes easiest to write them to a stream and retrieve them as a way of making a copy. There will be a big performance hit but sometimes that doesn't matter.

It's also worth noting that ParcPlace has changed default implementation of copy. Hans-Martin Mosner (hmm@heeg.de) writes:

> **In R4.1, the only copy method besides #shallowCopy is #copy itself. It is implemented as ^self shallowCopy postCopy. The postCopy method is the one that should do the dirty work. It can copy instance variables, leave others alone, update backpointers, and so on. Since it executes in the already copied object, it has access to everything it needs. To make copies which don't share instance variables, the postCopy methods should copy all such variables.**

This is a nice implementation, since postCopy doesn't need to do anything for variables that only require a shallow copy. Thus, adding instance variables doesn't necessarily require changing the copy method. My only complaint is that this change was not very well advertised; I only discovered it by stumbling across the code while doing something else.

INHERITANCE

As if there weren't already enough problems with copying, there also can be problems inheriting from a class that defines its own copying methods. For example, Ralf Grohman (ralf@ubka.uni-karlsruhe.de) writes:

> **I want to extend the Dictionary Class in some way. So I generated a new class (Test) which is a subclass of Dictio-**

nary and added an instance variable 'temp'. The sole method of this class is:

```
addiere
 temp := 'ok'.
 1 to: 5 do: [:x |
Transcript show: 'temp='; show: temp printString; cr.
   self at: x put: 'Test'. ].
```

When I call it via **'Test new addiere.'** I get the following Transcript:

```
temp= 'ok'
temp= 'ok'
temp= nil
temp= nil
temp= nil
```

Hey! Why is the instance variable overwritten after the second iteration?

This problem is ParcPlace-specific and is described by Rick Klement (rick@rick.infoserv.com):

> It was not overwritten. It just wasn't moved to the new object created when the **Dictionary** had to grow to accommodate three entries. Welcome to one of Smalltalk's more subtle bugs. . . . I'll bet this bug exists in 10% of the large programs that add instance variables to variable classes.

ParcPlace Smalltalk implements classes such as Dictionary, Set, and OrderedCollections as variable classes (classes with indexed instance variables). When instances need to grow, a new, larger instance is created, and become: is used to replace the old collection with the new. Unfortunately, the grow method only copies the indexed instance variables. If non-indexed instance variables are present they must be copied explicitly, and user classes must override the grow method to do this. Jan Steinman (steinman@hasler.ascom.ch) writes:

> There have been many debates about how to best handle this. One might be an off-line "checker" method that would look for **SequenceableCollection** subclasses that add instance variables but do not implement **#grow.** I once reimplemented **#grow** so that it copied all instance variables, rather than specific ones (**1 to: self class instSize do: [:i | ...]**). But this gets you into trouble in some cases where the new **Collection** requires different values, such as **'firstIndex'** and **'lastIndex'** in **OrderedCollection**. . . . For the time being, the answer is to make sure people understand what is happening, but I've been Smalltalking for eight years, and it still bites me now and then!

Another possible solution is to implement collections differently. In Digitalk's version, these are normal classes that have an array as an instance variable. If the collection needs to grow, then a larger array is created, its contents are copied, and the instance variable replaced. It requires an extra layer of indirection for collection access, but become: is not necessary and the instance variables don't need to be copied. Digitalk's reason for doing this is probably that become: is a very expensive operation in their dialects, but Ralph Johnson (johnson@cs.uiuc.edu) argues that this is a cleaner implementation. In fact, he has code to change Smalltalk-80 to operate this way:

> I have a **fileIn** that will do this to 2.3, but haven't got around to doing it to any of the later images. You can't change classes like **MethodDictionary**, of course, but you can eliminate most of the old-style collections.

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He currently works in ParcPlace Smalltalk on problems relating to finite element analysis and has worked in most Smalltalk dialects at one time or another. He can be reached at +1.613.788.2600 x5783.

GETTING REAL

Juanita Ewing

Constants, defaults and reusability

This column focuses on two aspects of reusability—subclassing and client usage—and how they relate to constants and defaults. Many classes have constants and defaults to represent commonly used values. Some of the values represented as constants may not really be constants, such as heuristically determined values, which are often hard-coded and embedded into methods. Though expedient in the prototyping stage, most constants should evolve into defaults as classes are refined. Developers of reusable software need to create reasonable defaults and include a mechanism to override them.

This column will show you how to use constants and defaults and still maintain a high level of reusability. We will examine several classes and methods from the Windows and OS/2 versions of Smalltalk/V that contain defaults. We will also revise some existing image code that has embedded constants and improve its reusability.

CONSTANTS

Many initialization methods contain constants and their values are often Smalltalk literals. In the class EntryField, the initialize method contains four constants: a string, an integer, a point, and a boolean. An initialization method is an appropriate place for constants. Subclasses typically override the initialize method to customize initial values:

```
initialize
    "Private - Initialize the receiver."
    value := ''.
    maxSize := 32.
    selection := 1@1.
    modified := false.
    ^super initialize
```

A less appropriate location for constants is embedded in arbitrary methods. A method should have one purpose: to define a default or perform some computation, but not both. With an embedded constant, reusability is impacted because it is difficult to:

- find and modify the constant
- override the constant in a subclass

The method file: in DiskBrowser has a constant that controls file contents display based on the file size. This constant is a size limit used to determine whether to display the entire file or a portion of it. If the file size exceeds this limit, it takes an extra action to see the entire contents. The main purpose of the file: method is to display the file contents. It should not contain the definition of the size limit:.

```
file: filePane
    "Private - Set the selected file to the selected one in filePane. Display
the file contents in the text pane."
    | aFileStream |
    CursorManager execute change.
    self changed: #directorySort:.
    selectedFile := filePane selectedItem.
    self switchToFilePane.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    wholeFileRequest := aFileStream size < 10000.
    aFileStream close.
    wholeFileRequest
        ifTrue: [self fileContents: contentsPane]
        ifFalse: [self showPartialFile]
```

Another DiskBrowser method, showPartialFile, also contains this constant. Having the same embedded constant in two methods can lead to maintenance problems:

```
showPartialFile
    "Private - Display the head and tail of the selected file in the text
pane."
    | aFileStream fileHead fileTail startMessage endMessage cr |
    CursorManager execute change.
    contentsPane modified: false.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    cr := String with: Cr with: Lf.
    startMessage := 'File size is greater than 10000 bytes, ',cr,
'first 1000 bytes are ...', cr.
    endMessage := cr, '************************', cr,
'last 9000 bytes are...', cr.
    fileHead := aFileStream copyFrom: 1 to: 1000.
    fileTail := aFileStream
        copyFrom: aFileStream size - 9000
        to: aFileStream size.
    aFileStream close.
    contentsPane
        fileInFrom: (ReadStream on: (startMessage,
fileHead,endMessage, fileTail));
        forceSelectionOntoDisplay.
    (self menuWindow menuTitled: '&Files') enableItem:
#loadEntireFile.
    (self menuWindow menuTitled: '&File') disableItem: #accept.
    CursorManager normal change
```

DEFAULTS

Developers should not embed constants in arbitrary methods.

Instead, each constant should be defined in a separate method, allowing it to be easily identified and overridden. Once isolated, we call these values defaults because subclasses easily can override the defining method, increasing the reusability of the class.

The method initWindowSize, from the class WindowDialog, specifies the initial size of a dialog. Because this value is isolated in a method, we consider it a default—subclasses easily can override the default initial window size:

```
initWindowSize
    "Private-Answer the default window size."
    ^150 @ 100
```

Another example from the image involves the application framework class ViewManager. The class ViewManager has a method that specifies the class of the top pane in the view structure. Subclasses easily can override this method to specify another top pane class, giving subclasses the critical ability to override the creation of collaborators:

```
topPaneClass
    "Private-Answer the default top pane class."
    ^TopPane
```

EVOLVING CONSTANTS INTO DEFAULTS

In the section above, we saw two DiskBrowser methods containing an embedded constant, 10000. Next we see the two original methods rewritten, plus one other method that isolates the file size limit for automatic reading. The isolated constant is now a default because it easily can be overridden by subclasses. With a default, maintainers can locate the limit more easily and are less likely to create inconsistent methods caused by modifying one but not the other reference to the constant:

```
autoReadLimit
    "Return the file size limit that determines whether the entire contents
    of a file will be automatically displayed."
    ^10000

file: filePane
    "Private - Set the selected file to the selected one in filePane. Display
    the file contents in the text pane."
    | aFileStream |
    CursorManager execute change.
    self changed: #directorySort:.
    selectedFile := filePane selectedItem.
    self switchToFilePane.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    wholeFileRequest := aFileStream size < self autoReadLimit.
    aFileStream close.
    wholeFileRequest
        ifTrue: [self fileContents: contentsPane]
        ifFalse: [self showPartialFile]

showPartialFile
    "Private - Display the head and tail of the selected file in the text
    pane."
    | aFileStream fileHead fileTail startMessage endMessagecr limit
    initial final |
    CursorManager execute change.
    limit := self autoReadLimit.
    initial := limit // 10 roundTo: 1000.
    final := limit - initial.
    contentsPane modified: false.
    aFileStream := selectedDirectory fileReadOnly: selectedFile.
    cr := String with: Cr with: Lf.
    startMessage :=
        'File size is greater than ', limit printString, ' bytes, ',cr,
        'first ', initial printString, ' bytes are ...', cr.
    endMessage :=
        cr, '**********', cr,
        'last ', final printString, ' bytes are ...', cr.
    fileHead := aFileStream copyFrom: 1 to: initial.
    fileTail := aFileStream
        copyFrom: aFileStream size - final
        to: aFileStream size.
    aFileStream close.
    contentsPane
        fileInFrom: (ReadStream on: (startMessage,fileHead,
        endMessage, fileTail));
        forceSelectionOntoDisplay.
    (self menuWindow menuTitled: '&Files')
    enableItem: #loadEntireFile.
    (self menuWindow menuTitled: '&File') disableItem: #accept.
    CursorManager normal change
```

INSTANCES MODIFY DEFAULTS

In addition to allowing subclasses to override defaults, developers can structure code so that instances can modify the default, improving client reuse. In this scenario, the class provides:

- storage for the default value, usually an instance variable
- accessing method for setting the default
- accessing method for retrieving the default (optional)

The class EntryField has a default for the maximum number of characters in an instance of EntryField. In addition to the initialize method we saw above and an instance variable to hold the value, one other method accesses the default maxSize. The accessing method maxSize: allows instances to customize the maximum number of characters that can be typed in an EntryField.

```
maxSize: anInteger
    "Set the maximum number of characters in the receiver to an Integer."
    maxSize := anInteger.
    handle = NullHandle
    ifFalse: [ self setTextLimit ]
```

There are several ways to provide an initial value for a default. In the initialize method for EntryField, maxSize is set to 32. An alternative design, shown below, has an accessing method that provides a default. The initialize method no longer sets the value of maxSize. In this case, the initial default value is only used if the default has not been otherwise set:

```
maxSize
    "Return the maximum number of characters that can be entered in
    the receiver. If no other value has been set, use the initial max size
    value and remember it."
    maxSize == nil
        ifTrue: [maxSize := self initialMaxSize].
```

continued on page 16

GUIs

Greg Hendley and Eric Smith

A quick look at two interface builders

In this installment of GUI Smalltalk, we will look at Smalltalk's two main interface builders: Cooper & Peters' WindowBuilder for different dialects of Smalltalk/V and ParcPlace's VisualWorks in R4.

While most people would not choose their Smalltalk dialect based on the interface builders available for it, it is interesting as a user and creator of graphical user interfaces (GUIs) to compare tools and see how two providers make use of GUIs themselves.

APPLES VS. APPLES OR ORANGES

The first question in comparing WindowBuilder and VisualWorks is "Are we comparing apples and apples or apples and oranges?" The answer is apples and apples. First, both are interface builders, not application builders; as such, their power is in graphically laying out the subpanes (if you are from V), controls (if you are from PM), or visualComponents (if you are from R4) of a window. This eliminates the need for you to calculate and write framing blocks.

COMPATIBILITY WITH THE ICM FRAMEWORK

Both WindowBuilder and VisualWorks output one class per window that can be used as the interface layer of the ICM framework. (The ICM framework was described in two previous installments of this column.) In WindowBuilder the default superclass of the output class is ViewManager. In VisualWorks the default superclass is ApplicationModel.

CAPABILITIES FOR CREATING USER INTERFACES

Similar capabilities

The capabilities of the two interface builders are more similar than different. Both have various versions of buttons, lists, static text, text editors, graphics, etc., and both help you build and test menus.

Sizing, positioning, and resizing of the window and its elements (subcomponents or subpanes) are supported in both. Elements of the user interface can be told to initially have the same width or height. They can be aligned like text: justified left, right, top, or bottom; centered horizontally; or centered vertically with respect to each other. Each element can be resized by absolute position or by ratios.

Both interface builders provide support for specifying each element's response to user input; both provide direct access to elements through the use of identifiers; both support tabbing; and, most important, both allow for the use of custom subpanes and visual components.

Differences in capabilities

Four capability differences between the two interface builders are noted below. Some are differences in degree while others appear in one but not the other; these include keyboard shortcuts, reuse, specifying response to user input, and specifying dependencies between components.

WindowBuilder provides direct support for keyboard shortcuts for menu items. VisualWorks does not provide such support from their tools.

VisualWorks provides support for three levels of user interface reuse. A user interface can be parameterized to work with any of a number of models. Inheritance can be used to let a subclass add visual components to its superclass. One interface can be used as a component in another interface. WindowBuilder only supports parameterization to use any number of models.

WindowBuilder provides support for specifying response to many types of user input. WindowBuilder tells you the events that may occur, lets you type the name of the method to invoke, and writes a stub for the method. For example, you can specify how to get the list for a list pane and what to do when a selection is made in the list pane. VisualWorks provides direct support only for specifying how to get the list. Responding to selection has to be explicitly coded in VisualWorks.

VisualWorks directly supports dependencies between different visual components in the same window. By making more than one component interested in a single aspect, all components respond when that aspect changes. Such dependencies have to be explicitly coded in WindowBuilder.

THEIR OWN USE OF USER INTERFACE TECHNIQUES

It is always interesting to see how the creators of an interface builder choose to use their tool. Let's start with their similarities.

Similarities

Both interface builders:

- operate in build-only mode

- provide buttons for committing the interface to code and for launching the interface
- use palettes that allow you to lay out panes or components as if you were using a drawing tool
- will open a browser on the generated code

One vs. many windows

The most obvious user interface difference between the two interface builders is the windows they use. WindowBuilder uses a single window with dialogs as needed. When a dialog is open the main window may not be used until that dialog is dismissed. VisualWorks makes use of a multitude of windows simultaneously, which some people call outboard windows. The window being built (the canvas) is in one window, and the outboard windows all operate on the canvas. Most, but not all, outboards operate on the most recently selected canvas.

Both techniques (outboards and dialogs) address the issue of clutter. The outboards allow users to decide how much information they want to see at once. However, this comes at a price. The canvas and the outboards are not visually tied together; it is not always clear which windows go together in VisualWorks, or even which windows are part of VisualWorks.

Resizing control

In both interface builders, the window being built responds to changes in the framing parameters of its panes or components. If a pane or component is given ratios instead of absolute positions, that pane changes shape as you change the initial size of the main window. WindowBuilder goes a step further and provides before and after silhouettes of your pane. As you change the framing parameters for a pane, it shows you a silhouette of your pane in the current window dimensions and also shows you the dimensions of your pane in a larger, resized window. This way WindowBuilder gives you immediate feedback.

SUMMARY

The two interface builders are more similar than different. The most important similarity is that they both fit nicely into the Interface part of the ICM framework, which lets you reuse design between dialects. After all, reuse of design is more powerful than reuse of code. ■

Greg Hendley is a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using various dialects of Smalltalk and various image generators. Eric Smith is also a member of the technical staff at Knowledge Systems Corporation. His specialty is custom graphical user interfaces using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corporation, 114 MacKenan Drive, Cary, NC 27511, or by phone, 919.481.4000.

■ **GETTING REAL** *continued from page 14*

```
    ^maxSize

initialMaxSize
    "Return the initial maximum size for text entry."
    ^32

initialize
    "Private - Initialize the receiver."
    value := ''.
    selection := 1@1.
    modified := false.
    ^super initialize
```

DEFAULTS REPLACE ARGUMENTS

Defaults also can be used to diminish interaction complexity. Commonly used values do not need to be passed as parameters; they can become defaults instead. Developers need to provide a way to override default values and still provide for the most common situations in which defaults are an applicable value.

The typical way for developers to provide default arguments is with additional methods that leave out key words. The method fill:rule:, from GraphicsTool, calls fill:rule:color: with the fill color set to the foreground color, which is a default. To override the default, the message fill:color:rule: can be sent:

```
fill: aRectangle rule: aRopConstant
    "Fill a Rectangle in the receiver medium with foreColor using
    aRopConstant."
    self fill: aRectangle rule: aRopConstant color: foreColor
```

CONCLUSION

The important difference between constants and defaults is their effect on reusability. Defaults, isolated in a method, are easily overridden by subclasses. Default values can be modified by instances if developers add enough support or can be used to eliminate arguments and reduce interaction complexity. Developers should always strive to evolve constants into defaults to make their classes more reusable. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services. She has been a project leader for several commercial O-O software projects and is an expert in the design and implementation of O-O applications, frameworks, and systems. In a previous position at Tektronix Inc., she was responsible for the development of class libraries for the first commercial-quality Smalltalk-80 system. She can be reached at 503.242.0725.

SMALLTALK IDIOMS

Kent Beck

A short introduction to pattern language

This will be a departure from my code-oriented columns. For the last six months I've been surreptitiously presenting my material using a technique that I've been working with for the past six years or so. This technique was derived from work done in architecture (buildings, not chips) to help people design comfortable spaces for themselves. The time has come to tell you what I've been leading up to, so that I can directly refer to these concepts in the future.

First, though, I have to tell you about the most thoroughly useful little idiom I have seen in a long time. Ward Cunningham and I recently got to code together on a nifty spreadsheet project and he showed me a simple idiom for dealing with nil values. It saves me a line in many methods and, since most methods are three or four lines long, that's a significant savings. Here is the implementation:

```
Object>>ifNil: aBlock
  ^self

UndefinedObject>>ifNil: aBlock
  ^aBlock value
```

Simple, huh? Here what happens when you use it, though. You can transform code that looks like:

```
foo isNil ifTrue: [foo := self computeFoo].
^foo
```

into:

```
^foo ifNil: [foo := self computeFoo]
```

The savings comes because ifTrue: and ifFalse: return nil if the receiver is false or true, respectively. IfNil: returns the receiver, which can be any object, instead. I have found ifNil: useful in many more situations than the one listed above. Try it! If you find a clever use, send it to me and I'll write it up.

The one complaint about ifNil: is that it is slower than "isNil ifTrue:" (or its grosser cousin "== nil ifTrue:"). I claim that if you are focused on anything but achieving the most readable code possible in the middle 80% of a development, you're doing the wrong thing. Besides, it wouldn't be that hard to implement ifNil: as an inline message, just like the other conditionals. If it's not that hard, maybe I should write it up some time. Or maybe you should!

Now back to our regularly scheduled column...

The problem to be solved is describing the intent behind a piece of code to someone who needs to use it. There are plenty of methods for describing how code works (even though most programmers aren't disciplined in using them), but describing how code is supposed to be used is a black art. As the emphasis on programming shifts from just running programs to refining and reusing them, this is a problem of increasing importance. As objects are supposed to be about reuse, describing intent is of critical importance to us.

Donald Knuth has attacked the problem with what he calls "Literate Programming." He shares the insight that programmers ought to write programs for other programmers, not just the computer. His solution is to make programs read like books. When you read a literate program you are reading a combination of prose and code. You can filter out the non-program elements and run the result through a compiler to get an executable program.

There are a couple of problems with literate programming as Knuth conceives it. First, his literate programming system is implemented as a 1970s–style textual language. To write a literate program you have to know the programming language, the typesetting language, and the extensions required by the literate programming system. More importantly, the structure of a literate program is fundamentally linear. It is intended to be read from beginning to end. While this may be appropriate for a monolithic program like TeX, it does not address the problem of describing the intent of an object library, which is intended to be used piecemeal—sometimes just by instantiating objects, sometimes by plugging new objects into existing frameworks, and sometimes by refinement.

What we need is a structure for intention-oriented information that is flexible enough to convey a variety of information at different levels, but structured enough to provide a predictable experience for readers. It has to be able to convey process-oriented information but also describe programs piecemeal. It has to describe both how a program is intended to be used and how it works.

The solution I have been pursuing derives from the work of architect Christopher Alexander, who has spent many years seeking a way for architects to describe generic solutions to architectural problems so that individuals can adapt these solutions to their situations. The solution he found, called pattern language, solves all of the problems listed above: It is piecemeal, but also has large-scale structure; its essence describes the application of a solution, but also relates how the solution works; and it describes solutions at all scales, from urban planning to the size and color of trim in a house. His approach is presented in a pair of books from Oxford Press: THE TIMELESS WAY OF BUILDING and A PATTERN LANGUAGE.

PATTERNS

The unit of knowledge in a pattern language is a pattern. A pattern encodes an adequate solution to a problem known to arise in the process of building a system. A person should be

able to read a pattern and know :

- what problems need to be solved before this one can be solved
- what problem the pattern solves
- what constrains the solution to the problem
- what to do to the system to satisfy the pattern
- what problems to solve once this one has been solved

Patterns have a consistent structure. Each has the following sections:

- a name evoking the problem and its solution
- a prologue summarizing what other patterns have to be considered before this one is appropriate
- a one-paragraph preamble describing the crux of the problem solved by the pattern
- a diagram illustrating the problem
- a short essay exploring constraints on the solution
- one or two paragraphs describing how to solve the problem
- an illustration of the solution
- an epilogue summarizing patterns that can be considered once this one is satisfied

Several valuable traits are common to all patterns:

- They always call for concrete actions, even if they are at very high levels. For instance, a design-level pattern might call for splitting one object into two to improve flexibility. A coding pattern might help you give names to arguments.
- They include a complete description of the considerations influencing the solution. Almost no documentation describes the forces acting on a decision, but it is precisely this information that allows you to evaluate an object for usefulness in a particular context.
- They are illustrated with a simple diagram. Alexander's patterns are remarkable for the degree to which their essence can be distilled into a simple line drawing. The effective computer patterns I have discovered also boil down to a little picture.

The word "pattern" takes several meanings in this context. First, each solution represents a pattern of elements. The object that uses an OrderedCollection has a specific relationship with the objects it references. Second, the constraints acting on the solution form a pattern. The need to conserve space tugs this way, the desire for greater speed that way. Finally, and most curiously, are common patterns of human behavior. The act of choosing an OrderedCollection recurs many times and in many places.

PATTERN LANGUAGE

Patterns do not stand in isolation. The epilogue and prologue sections of each pattern link it to several others. The result can be seen as a kind of lattice, with problems that need to be addressed first higher than those that can be considered later. Much of an expert's skill comes from knowing what to worry about up front and what can be safely postponed. This process-oriented information is often as valuable as the patterns themselves.

The patterns together form a language in the sense that the patterns are terminal symbols, and the links between them are the productions. You create well-formed sentences by considering a sequence of patterns in turn. The result is a fully formed system. This is the primary difference between a pattern language and a set of design rules (like the Apple Human Interface Guidelines). The pattern language helps you create a system with the desired properties, not just analyze existing systems for the existence of those properties. A pattern language for good design will lead you to create a system with high coherence and low cohesion, not just describe the properties in isolation.

A complete pattern language for object-oriented programming encompasses patterns at all levels. Broad patterns cover issues like distribution of responsibility and control structures. Subsequent patterns help use the right abstractions in a library. Final patterns deal with variable naming, method naming, breaking methods into smaller methods, factoring code into inheritance hierarchies, and performance tuning.

CONCLUSION

No one has yet written a pattern language for objects like the one outlined above. There is general agreement that the problem of communicating intent is critical to cashing in on the promise of object-oriented programming. Researchers worldwide have turned to pattern languages as a promising approach to the problem. Here are a few I know about:

- Ralph Johnson at the University of Illinois is writing a pattern language for Hot Draw, a graphical editing framework.
- Richard Helm and John Vlissides of IBM and Erich Gamma of the Union Bank of Switzerland have been writing a catalog of "design patterns," which capture common design elements of C++ programs.
- Bruce Anderson of the University of Essex is leading an effort to compile an "architecture handbook."
- Oscar Nierstrasz at the University of Geneva has been using patterns to try to achieve reuse.

In subsequent columns I will explicitly use the pattern format where appropriate to describe Smalltalk idioms.I recommend the study of Christopher Alexander's work for those interested in attacking the educational side of the reuse problem. I have enjoyed studying the material both because of the obvious parallels between the pitfalls of professional architects and professional programmers, and because I am now far more sensitive to my physical environment Iand its effect on my life.

Architecture has the advantage (and disadvantage) of thousands of years of history to mine for patterns. Programming is a new enough discipline that we all have to invent new solutions often. Collecting and disseminating these common patterns will hasten the day we can get on to more interesting questions. As you discover patterns in your own work please send them to me. ■

Kent Beck has been discovering Smalltalk idioms for eight years at Tektronix, Apple Computer, and MasPar Computer. He is also the founder of First Class Software, which develops and distributes reengineering products for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226, by phone at 408.338.4649, fax 408.338.3666, or compuserve 70761,1216.

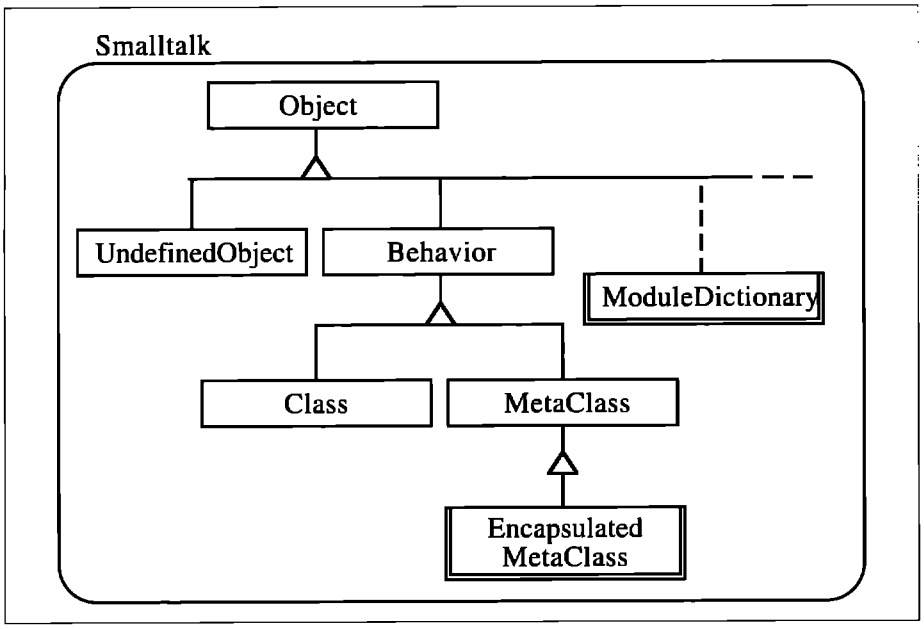


Figure 5.

ADDING MODULES TO SMALLTALK

Where a normal Smalltalk class uses a Dictionary for its pool of class variables, a module class uses a ModuleDictionary for its domain. The ModuleDictionary class is similar to the SystemDictionary class. Like the Smalltalk system dictionary, each module domain can contain shared objects, including other Smalltalk classes. In addition, each module domain keeps track of the names of the module class variables.

Each class contained in a module domain needs to know what module contains it. For this reason, each class contained inside a module domain is associated with an EncapsulatedMetaClass rather than a MetaClass. The class EncapsulatedMetaClass extends the class MetaClass by adding a reference to the module whose domain contains the encapsulated class.

Figure 5 depicts the classes changed to extend Smalltalk/V. Rectangles with doubled borders indicate the new classes.

RESOLVING SHARED NAMES

Smalltalk methods use names that start with lower case for private names, including instance variable names, method arguments, and block temporaries. Smalltalk methods also can reference shared objects whose names are capitalized.

The visibility of these shared names depends on where they are located in the system. Shared names can be found in class variable pools, global pool dictionaries, and the Smalltalk system dictionary. During method compilation, references to shared names are resolved by searching dictionaries in the following order:

- class variable pools of the class and its superclasses up through the class Object
- pool dictionaries to which the class subscribes from the Smalltalk system dictionary
- the Smalltalk system dictionary itself

Extending the visibility rules of the compiler is the key to adding modules to Smalltalk. The Smalltalk system dictionary is the enclosing domain for classes not contained in a module. As such, it is also considered the system domain. Because a module contains a name space in its domain, references to shared names are resolved by searching dictionaries in the following order:

- class variable pools of the class and its superclasses up through the class Object
- pool dictionaries to which the class subscribes in the module domains enclosing the class up through the Smalltalk system domain
- module domains enclosing the class up through the Smalltalk system domain

Listing 1.

ClassFiler objects are responsbile for filing Smalltalk source code in and out of streams, usually FileStreams. This example is derived from the Smalltalk ClassReader. It shows how private methods can be encapsulated in a module.

```
"The public interface module class."
Object moduleSubclass: #ClassFiler
  instanceVariableNames:
    'privateSelf '
  classVariableNames: ''
  poolDictionaries: '' !

"The private ClassFiler class."
Object subclass: #ClassFiler in: ClassFiler
  instanceVariableNames:
    'class '
  classVariableNames: ''
  poolDictionaries: '' !

!ClassFiler class methods !
forClass: aClass
    "Answer a new instance of a public ClassFiler
     object."
    ^self new forClass: aClass! !

!ClassFiler methods !
fileInFrom: aStream
    "Read chunks from aStream. Compile each
     chunk as a method for the class described
     by the receiver. Log the source code of the
     method to the change log."
    | stream |
    stream := Sources at: 2.
    stream setToEnd.
    privateSelf instanceHeaderOn: stream.
    privateSelf fileInFrom: aStream.
    stream nextChunkPut: ''; flush!
fileOut: methodName On: aStream
    "File out the named method for the class
     described by the receiver to aStream, in
     chunk format."
    privateSelf checkFor: methodName.
    aStream cr.
    privateSelf instanceHeaderOn: aStream.
    privateSelf fileOut: methodName On: aStream.
    aStream nextChunkPut: ''; cr.!
fileOutOn: aStream
    "File out all the methods for the class described
     by the receiver to aStream, in chunk format."
    aStream cr.
    privateSelf instanceHeaderOn: aStream.
    privateSelf fileOutMethodsOn: aStream.
    aStream nextChunkPut: ''; cr!
forClass: aClass
    "Answer the receiver after attaching a new
     private instance of the private ClassFiler class."
    privateSelf := ClassFiler new setClass: aClass.! !

!ClassFiler ClassFiler class methods ! !

!ClassFiler ClassFiler methods !
checkFor: methodName
    "Verify that the class described by the receiver
     contains the named method."
    class methodDictionary
        at: methodName
        ifAbsent: [
            ^self error:
                methodName asString,
                ' is missing from ',
                class printString
        ].!
fileInFrom: aStream
    "Read chunks from aStream until an empty
     chunk (a single bang '!!') is found. Compile each
     chunk as a method for the class described
     by the receiver."
    | aString result |
    [   aString := aStream nextChunk.
        aString isEmpty
    ]
    whileFalse: [
        result := class compile: aString.
        result notNil ifTrue: [
            result value sourceString: aString
        ]
    ].!
fileOut: methodName On: aStream
    "File out the named method for the class
     described by the receiver on aStream, in
     chunk format."
    aStream cr; nextChunkPut: (
        class sourceCodeAt: methodName
    ).!
fileOutMethodsOn: aStream
    "File out all of the methods for the class
     described by the receiver on aStream."
    class selectors asSortedCollection do: [ :selector |
        self fileOut: selector On: aStream
    ].!
instanceHeaderOn: aStream
    "Write a header which identifies the class
     described by the receiver on aStream."

    "Note that filing in translates double bangs to
     single bangs and filing out translates single
     bangs into double bangs (like those used here)."

    aStream
        cr; nextPut: $!!;
        nextPutAll: class printString;
        space; nextPutAll: 'methods !!'!
```

Because these new visibility rules subsume existing rules, the semantics of normal classes continue to be supported.

BREAKING AND ENFORCING MODULE ENCAPSULATION

Because modules enclose and encapsulate their private classes, programming tools need a way to break the encapsulation of the module to create new classes inside the module. For this reason, a change has been made to class Class.

When a module class sends #doesNotUnderstand: aMessage, the message selector is checked to see if it is a capitalized unary selector that is the name of a private class inside the module. If so, the message answers the requested private class from the module. Otherwise, the message is dealt with using the existing #doesNotUnderstand: behavior.

This revised behavior is provided expressly for the compiler and development tools. This service breaks the encapsulation of the module similar to the way #instVarAt: breaks the encapsulation of an object.

To enforce the encapsulation of a finished module, the module can be closed by adding another version of #doesNotUnderstand: to the module class, overriding the one in class Class. This can be accomplished simply by sending the message #closeModule to the module class:

```
ModuleA closeModule.
```

This forces other classes outside the module scope to use the publicly defined interface to the module.

MODULE INTERFACES

The module that encloses a group of private classes can provide either direct or indirect access to the services of those classes. If

the module grants direct access to an enclosed class by publishing it, then all the services of that class are directly available.

A module can provide direct access to an enclosed private class by supplying an accessing message as part of the public interface to the module. Suppose we want to give direct access to SubclassB in Figure 3. We could give ModuleA a class method named #SubclassB that answers SubclassB:

```
!ModuleA class methods !
SubclassB
    "Publish SubclassB."
    ^SubclassB! !
```

However, modules provide their greatest advantage when they hide or limit the visibility of their internals. This visibility is determined by what information (objects) is revealed by the module class and its instances (if any). The module forms the public interface to the classes inside the module domain.

> “Extending the visibility rules of the compiler is the key to adding modules to Smalltalk.”

COMPARISONS WITH OTHER WORK

Several other works [3,5,6] suggest that modules are not first-class and have no direct representation in an active system of objects. They suggest that modules only serve as name spaces for controlling the visibility of shared names. This article has presented a different viewpoint, advocating the inclusion of modules as a special kind of class.

Using a responsibility-driven approach,[5,7] the design of an object system can achieve a high degree of encapsulation and reusability. Classes help to maintain encapsulation when they limit access to their variables. Modules can help to maintain a higher degree of encapsulation by limiting access to the private behavior of subsystems.

The Law of Demeter[8] suggests that object systems can best realize the benefits of reuse by strictly limiting the visibility of objects to those other objects in the system that require such visibility. With classes and modules, visibility is controlled by the system designer.

CONCLUSION

This article shows how modules can be made first-class within Smalltalk systems. Modules provide a natural way of packaging object systems and give object system designers more options for controlling the visibility of a system's implementation details. Modules reduce the possibility of naming conflicts between separable systems of objects.

Just as classes form a hierarchy for the inheritance of structure and behavior, modules can be used to form a nested hierarchy of name spaces (domains). The organizing principles of classes and modules are orthogonal and complement each other.

Classes can be imported into modules by adding a private subclass of the same name to the module domain. However, given the new visibility rules for shared names, this kind of transparent subclassing may be the only reason for explicitly importing classes from outside a module.

Classes can be exported from a module by providing a message for accessing the class by name. However, this kind of revelation on the part of a module is discouraged because it leads to dependencies on the module's internals.

SOURCE CODE AVAILABILITY

Modules may be added to Smalltalk with relatively few changes. Two new classes and some changes to various core Smalltalk classes and the front end of the compiler provide the essentials for creating module classes. A tool for browsing module domains is included. This shows one way that support for modules may be integrated into the programming tools. The source code for adding modules to Smalltalk/V is available through the American Information Exchange (AMIX). ■

References

1 Wirth, N. PROGRAMMING IN MODULA-2, TEXTS AND MONOGRAPHS IN COMPUTER SCIENCE, 2nd Edition, David Gries, Springer-Verlag, Berlin, 1984.

2 Booch, G. SOFTWARE ENGINEERING WITH ADA, Benjamin/Cummings, Menlo Park, CA, 1983.

3 Wirfs-Brock, A. and B. Wilkerson. An overview of modular Smalltalk. OOPSLA 1988 PROCEEDINGS, September 1988, pp. 123-134.

4 Rumbaugh, J. et al. OBJECT-ORIENTED MODELING AND DESIGN, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

5 Wirfs-Brock, R. B. Wilkerson, and L. Wiener. DESIGNING OBJECT-ORIENTED SOFTWARE, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.

6 Szyperski, C.A. Import is not inheritance, why we need both: modules and classes. ECOOP 1992 PROCEEDINGS, June/July 1992, pp. 19-32.

7 Wirfs-Brock, R. and B. Wilkerson. Object-oriented design: a responsibility-driven approach. OOPSLA 1989 PROCEEDINGS, October 1989, pp. 71-75.

8 Lieberherr, K.L. and I. Holland. Formulations and benefits of the law of Demeter. SIGPLAN NOTICES, v24#3, March 1989, pp. 67-78.

Nik Boyd has been developing object systems since 1987. Since January 1990, he has been with Citicorp Transaction Technology Inc. in Santa Monica, California, where he is currently a Principal Member of the Technical Staff. His experience with OOP includes work with PARTS Workbench, Smalltalk/V for PM, Mac, Windows, and DOS, and Objectworks/Smalltalk v2.5 for DOS and v4.0 for Windows. His research interests include instance-centered and class-centered object systems, as well as tools and techniques that support object-oriented software engineering. Nik may be contacted via internet e-mail at 74170.2/71 @ CompuServe.com or through the American Information Exchange (AMIX).

PRODUCT ANNOUNCEMENTS

Product Announcements are not reviews. They are abstracted from press releases provided by vendors, and no endorsement is implied. Vendors interested in being included in this feature should send press releases to our editorial offfices, Product Announcements Dept., 91 Second Ave., Ottawa, Ontario K1S 2H4, Canada.

The Smalltalk Interface to Objective-C makes Objective-C objects look like Smalltalk objects. The interface is based on the simple concept that every remote Objective-C object can be represented by a local Smalltalk proxy object and every Objective-C class can be represented by a Smalltalk instance. Messages sent to a local Smalltalk proxy object are transparently forwarded to the actual Objective-C object it represents and the results are returned as Smalltalk objects. If the return value is an object ID, a proxy for that object is returned so that follow-on messages are also forwarded.

To the Smalltalk developer, there are just Smalltalk messages being sent to Smalltalk objects. To the Objective-C developer, there are just Objective-C messages being sent to Objective-C objects. The net result is that the two languages are very smoothly integrated. Developers no longer have to choose between using Objective-C or Smalltalk. They can use both languages together, each where it is best suited.

Berkeley Productivity Group, 35032 Maidstone Court, Newark, CA 94560, 510.795.6086, fax: 510.795.8077

The Object People Inc., a leading international provider of training, mentoring, and project development services in object-oriented technology, has expanded its educational facilities and launched a new internship program for Smalltalk programmers. The company specializes in the design and development of custom Smalltalk applications.

The new training facility allows the firm to offer an expanded schedule of open enrollment courses in Smalltalk/V, Objectworks\Smalltalk, and object-oriented concepts, analysis and design. In addition, the firm's Objectworks\Smalltalk courses now include the new VisualWorks application development environment recently introduced by ParcPlace. The Object People is also offering courses in PARTS, Digitalk's new "visual development tool" for OS/2.

The new internship program is designed to fast-track the development of accomplished Smalltalk programmers. Interns will have the opportunity to work on their own applications while having immediate access to assistance and guidance from experienced Smalltalk developers. Internships are flexible in duration and are spent at The Object People's educational facility in Ottawa. The program is available to both Smalltalk/V and Objectworks\Smalltalk developers. Participation in the program is strictly limited in view of the intensive one-on-one interaction required to make the program successful.

The Object People Inc., 509-885 Meadowlands Dr., Ottawa, Ontario, Canada, K2C 3N2, 613.230.6897, fax: 613.235.8256

Highlights

Excerpts from industry publications

DATABASES

. . . Is the decomposition of the Open OODB system into modules arbitrary, or will other efforts to build a system with similar functionality result in a similar factoring? It is too early to report that such experiments necessarily result in similar factorings, but the Open OODB's factoring into modules is very similar to the application integration framework being developed by the industrial consortium Object Management Group. . . . Thus, the OMG and the Open OODB architectures are almost isomorphic. It is interesting that one is viewed as an application integration framework architecture and the other as an OODB architecture. . .

Architecture of an open object-oriented database management system, David L. Wells, Jose/ A. Blakeley, and Craig W. Thompson,COMPUTER, 10/92

. . . The power of objects is in their robustness, extensibility, flexibility, and modularity. Actually I wish engineers did not have to know or care about objects. Except as interesting metaphors, they are not useful to any one but computer professionals. But we are not yet able to reach that level of information hiding. If you are selecting an engineering database management system today, it probably should be object-oriented—and if it isn't, you should know why not.

What's the big deal about objects?, Joel N. Orr, COMPUTER-AIDED ENGINEERING, 11/92

DESIGN

. . . Although it's nice that operating systems are becoming object-oriented for the user, there's no doubt that maintaining backward compatibility with a straight C API brings with it an

inherent complexity. Object management needs to be integrated much more smoothly into the operating system services and made to fit naturally with object-oriented languages. In effect, you want the operating system support for objects to be as transparent as support for memory allocation and deallocation, file services, and so on. The approach must be sufficiently general that it can accommodate a range of languages, not just C++ and Pascal. There will always be a place for interpreted languages such as Smalltalk and Actor, and I hope that future object-oriented operating systems will make cross-language sharing of objects a reality.

Polymorphism unbound, Zack Urlocker, WINDOWS TECH JOURNAL, 10/92

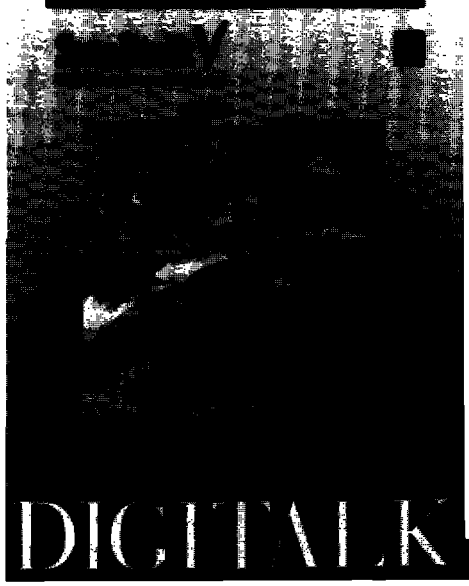
OOP is inclusive, just as structured programming was two decades ago. It differs, however, from structured programming's traditional association with functional design methods such as functional decomposition, dataflow diagrams or data structure design. In OOP, objects are first categorized into classes and organized hierarchically according to their dependency and similarity. Each class comprises a set of attributes reflecting the objects' generally static properties and a set of routines (in Smalltalk, methods) that manipulate these attributes. Then relations between classes, such as inheritance, are designed. . .

Object-oriented computing, David C. Rine and Bharat Bhargava, COMPUTER, 10/92

..."In the object world you start by defining classes," explained Lanny Lampl, a technical consultant in Levi Strauss" Information Resources Group. "You have to parcel out the responsibilities of each object and decide how classes will interact with each other." Carrying out an object-oriented analysis turned out to be harder than switching to SmallTalk. "The syntax of the language is not the big thing," Lampl said. "The important thing is learning how to think about objects."

Levi Strauss cuts client/server pattern, Jean S. Bozman, COMPUTERWORLD, 11/16/92

THE TOP NAME IN TRAINING IS ON THE BOTTOM OF THE BOX.



Where can you find the best in object-oriented training?

The same place you found the best in object-oriented products. At Digitalk, the creator of Smalltalk/V.

Whether you're launching a pilot project, modernizing legacy code, or developing a large scale application, nobody else can contribute such inside expertise. Training, design, consulting, prototyping, mentoring, custom engineering, and project planning. For Windows, OS/2 or Macintosh. Digitalk does it all.

ONE-STOP SHOPPING.

Only Digitalk offers you a complete solution. Including award-winning products, proven training and our arsenal of consulting services.

Which you can benefit from on-site, or at our training facilities in Oregon. Either way, you'll learn from a staff that literally wrote the book on object-oriented design (the internationally respected "Designing Object Oriented Software").

We know objects and Smalltalk/V inside out, because we've been developing real-world applications for years.

The result? You'll absorb the tips, techniques and strategies that immediately boost your productivity. You'll reduce your learning curve, and you'll meet or exceed your project expectations. All in a time frame you may now think impossible.

IMMEDIATE RESULTS.

Digitalk's training gives you practical information and techniques you can put to work immediately on your project. Just ask our clients like IBM, Bank of America, Progressive Insurance, Puget Power & Light, U.S. Sprint, plus many others. And Digitalk is one of only eight companies in IBM's International Alliance for AD/Cycle—IBM's software development strategy for the 1990's. For a full description and schedule of classes, call (800) 888-6892 x411.

Let the people who put the power in Smalltalk/V, help you get the most power out of it.

100% PURE OBJECT TRAINING.

DIGITALK