The Smalltalk Report

The International Newsletter for Smalltalk Programmers

October 1993 **Volume 3 Number 2**

PRAGMATIC MULTIPLE INHERITANCE

by Bob Beck

Contents:

THIS ARTICLE DISCUSSES pragmatic multiple inheritance (PMI), which provides a useful subset of full multiple inheritance functionality in Smalltalk. PMI does this without burdening the Smalltalk system with complex multiple-inheritance semantics or otherwise damaging the nice single-inheritance model of Smalltalk. PMI adds instance variables in target classes, which refer to instances of additional superclasses, and adds methods to forward messages to these instances. A new browser provides a user interface to support browsing and maintenance of PMI relationships.

Many cases where multiple inheritance is desired involve mixing-in properties of some sort. Mixing in a property implies adding this property to a class as if the class fully supports the messages and state of this property. PMI provides a means to mix-in properties in a class definition, where these property classes aren't appropriate as a unique superclass. While in some cases a single-inheritance class hierarchy can be refactored to avoid using the mix-in model, in many cases the result is neither reasonable nor intuitive. Further, even if the properties can be added to a single-inheritance class hierarchy, it is often difficult to arrange that only the desired properties are inherited, without picking up unnecessary (or undesired) properties in the process. PMI addresses these difficulties.

An example of such a property is *dirtiness:* a binary state that implies an object has been modified relative to a more permanent copy of that object. This is an often-used property (e.g., in a View to know if the state of the view has been modified from that of the model it represents), but is awkward to place correctly in a single-inheritance hierarchy. A common result is various forms of abstract DirtyXxx classes (e.g., DirtyView, DirtyModel), where each duplicates the dirtiness code.

The technique used in PMI can be done manually using existing Smalltalk browsing tools, but it is difficult to maintain and browse this way. The PMI browser provides an interface that makes the creation, maintenance, and removal of PMI easy to do.

PMI as discussed in this article has been implemented in VisualWorks v1.0. The source code for PMI has been placed in the public domain, and may be found via ftp in the Manchester/UI Smalltalk goodies archives (mushroom.cs.man.ac.uk (130.88.13.70) or st.cs.uiuc.edu (128.174.241.10)) as /pub/goodies/visual/pmi.st. These archives may also be reached via email by sending mail to goodies-lib@cs.man.ac.uk with the word *help* in the subject line.

DESCRIPTION

PMI is basically inheritance by delegation. It works by storing a reference to an instance of the PMI-superclass (the class being inherited from), and providing a means to forward messages to that instance. A class doesn't directly inherit from another class via PMI; rather, it includes an instance of that class among its instance variables.

PMI creates an instance variable in the new class (the PMI-subclass: The class inheriting from the PMI-superclass), which references an instance of the PMI-su-

continued on page 21...

John Pugh

Paul White

EDITORS' CORNER

Two of the articles featured this month focus on the issue of multiple inheritance. Inheritance, of course, is sold as one of the big features offered by object-oriented technology, and when used effectively allows for significant reductions in the amount of code needed to be written for a software project. There is little argument put forward as to the merit of inheritance as a development technique. Discussions abound, however, over whether multiple inheritance—the natural extension of single inheritance—is usable. "Why is it that C++'s got it, and Smalltalk doesn't?" "Well, Smalltalk could have it if it wanted it." "Well, why doesn't it?"

There will never be a definitive answer to these questions, since the answer always depends on context. It will always be possible to come up with examples where multiple inheritance is the way to go. String as a subclass of both Magnitude and Collection, and the implementation of ReadWriteStream are two obvious examples taken from the base Smalltalk class library. Similarly, one can always cite examples where the use of multiple inheritance is a disaster waiting to happen. Religious wars always have these examples.

At the heart of the question, though, is what we are actually trying to gain by using multiple inheritance (or inheritance for that matter!). If it's code reduction and reuse along with reduced maintenance costs, we find that managing single inheritance problems is difficult, let along multiple inheritance. The problem lies with the simple fact that objects change over time. As a particular class of object changes, or a family of objects in a hierarchy evolves, their characteristics and responsibilities change and so must their hierarchies. Pinpointing where these changes should be made, and how to do so effectively, is challenging and usually represents a considerable undertaking. Multiple inheritance only compounds the problem because the evolution of a class may impact others inheriting from it in ways that are not at all obvious or predictable; class relationships are now represented as a lattice rather than a tree.

So, is multiple inheritance a good thing? Well, perhaps, but realize that what you gain now may actually cost you much more to maintain in the future. . . .

These comments aside, the two articles presented in this month's issue present interesting discussions on how to best use inheritance. Bob Beck describes a means for attaining a limited form of multiple inheritance, where developers can explicitly "mix in" additional classes to an existing class. Bruce Samuelson discusses the implementation of a family of tree classes exhibiting a variety of properties, highlighting the design choices faced during such development.

Also in this issue, Ed Klimas returns with a pragmatic description of how Smalltalk can be used in the construction of real-time systems. Kent Beck presents a pattern for dealing with the tricky problem of better controlling what code gets inherited in a deep inheritance tree. Juanita Ewing explains how abstract classes can actually serve in tow different roles—as a design mechanism or as an implementation-based code-sharing mechanism. And finally, Alan Knight continues his exploration of extensions to the Smalltalk environment.

John Pugh Paul White

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 Inc., 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. For service on current subscriptions call 800.783.4903. Submit articles to the Editors at 509–885 Meadowlands Drive, Ottawa, Ontario K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f).
PRINTED IN THE UNITED STATES.

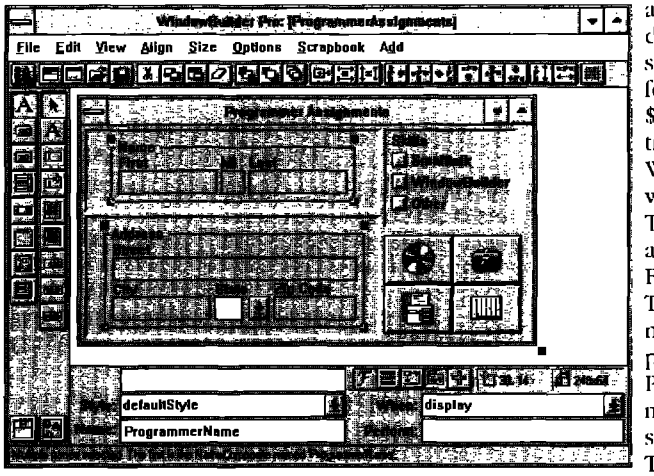
SHARE

OBJECT SYSTEMS INC.

WINDOWBUILDER PRO!

The New Power in Smalltalk/V Interface Development

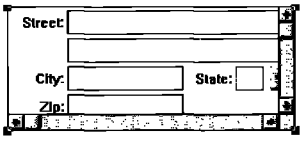
Smalltalk/V developers have come to rely on WindowBuilder as an essential tool for developing sophisticated user interfaces. Tedious hand coding of interfaces is replaced by interactive visual composition. Since its initial release, WindowBuilder has become the industry standard GUI development tool for the Smalltalk/V environment. Now Objectshare brings you a whole new level of capability with WindowBuilder Pro! New functionality and power abound in this next generation of WindowBuilder.



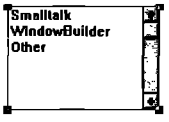
WindowBuilder Pro/V is available on Windows for $295 and OS/2 for $495. Our standard WindowBuilder/V is still available on Windows for $149.95 and OS/2 for $295. We offer full value trade-in for our WindowBuilder customers wanting to move up to Pro. These products are also available in ENVY®/*Developer* and Team/V™ compatible formats. As with all of our products, WindowBuilder Pro comes with a 30 day money back guarantee, full source code and no Run-Time fees.

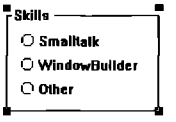
Some of the exciting new features...

• CompositePanes: Create custom controls as composites of other controls, treated as a single object, allowing the developer higher leverage of reusable widgets. CompositePanes can be used repeatedly and because they are Class based, they can be easily subclassed; changes in a CompositePane are reflected anywhere they are used.

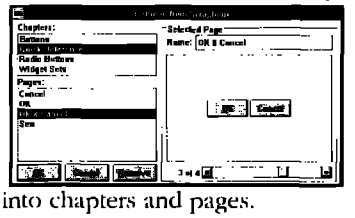


• Morphing: Allows the developer to quickly change from one type of control to another, allowing for powerful "what-if" style visual development. The flexibility allowed by morphing will greatly enhance productivity.

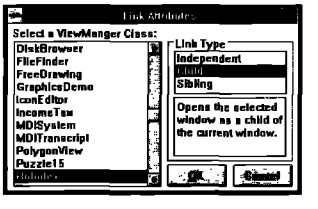


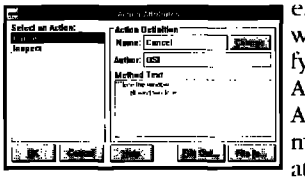


• ScrapBook: Another new feature to leverage visual component reuse, ScrapBooks provide a mechanism for developers to quickly store and retrieve predefined sets of components. The ScrapBook is a catalog of one's favorite interface components, organized into chapters and pages.



• Rapid Prototyping capabilities: With the new linking capabilities, a developer can rapidly prototype a functional interface without writing a single line of code. LinkButtons and LinkMenus provide a powerful mechanism for linking windows together and specifying flow of control. ActionButtons and ActionMenues provide a mechanism for developers to attach, create, and reuse actions without having to write code. These features greatly enhance productivity during prototyping.





• ToolBar: Developers can Create sophisticated toolbars

just like the ones in the WindowBuilder Pro tool itself.

• Other new features include: enhanced duplication and cut/paste functions, size and position indicators, enhanced framing specification, Parent-Child window relationship specification, enhanced EntryField with character and field level validation, and much more...

• Add-in Manager: Allows developers to easily integrate extensions into WindowBuilder Pro's open architecture.

Catch the excitement, go Pro!
Call Objectshare for more information.

(408) 727-3742

Objectshare Systems, Inc
Fax: (408) 727-6324
CompuServe 76436,1063

5 Town & Country Village
Suite 735
San Jose, CA 95128-2026

COPING WITH SINGLE INHERITANCE: BUILDING TREE CLASSES

Bruce Samuelson

How can you develop a class library using single inheritance for a problem domain that is most naturally modeled with multiple inheritance?

Smalltalk-80 included partial support for multiple inheritance in some older versions. Researchers have added experimental support to the current version. Digitalk has added some hooks in its current version for possible support in the future. However, industrial-strength support is not available in Smalltalk-80 or Smalltalk/V today.

The problem domain under consideration is tree structures. A tree is a hierarchical organization of nodes with intermediate branches and terminal leaves. Examples are file systems, hierarchical databases, parse trees, balanced trees, and the Smalltalk class hierarchy. The goal is to design tree classes that are robust, flexible, reusable, clear, simple, and reasonably efficient in space and time. The constraint is to do so within Smalltalk's single inheritance model.

The tree classes have four possible instance variables:

- Key uniquely identifies a node like a database key or a dictionary key
- Value stores data at the node
- SubTrees holds the immediate subnodes
- superTree points to the parent tree

Each needs a public read accessor, and all but one need a public write accessor. With one exception, each variable is optional and may be introduced independently of the others. This makes 2^4, or 16, possible classes. For example, for every class containing a superTree pointer, another class can be defined without one. Because a key cannot be defined apart from subTrees, four combinations are ruled out, resulting in 12 possible classes.

Where should support methods (validate, initialize, access, test, . . .) for an instance variable be placed? They can either be in each class defining the variable or in a common superclass. The only way to organize a single-inheritance class hierarchy that avoids duplication of support methods would be to place them in the top-level class. If multiple inheritance were available, they could be naturally defined in several superclasses.

THREE ATTEMPTS AT IMPLEMENTING TREE CLASSES

The adage is to throw the first one away if you want to get it right. I threw the first two versions away and am still not sure I got it right. Figure 1 shows the three design iterations. The third has the 12 classes resulting from possible combinations of instance variables. Later, I'll explain the last class, BinaryTree.

There were three problems with the first two design attempts. They lacked an abstract superclass analogous to Collection, making new application subclasses more difficult to write. They forced subclasses to inherit instance variables (key and subTrees) that the subclass might not need. And there were too many methods with nearly identical code.

DESIGN PRINCIPLES

The third iteration of tree classes worked better. It is based on familiar design principles. The single inheritance constraint often forced these principles to be pushed to their limit.

VALIDATE THE CLASSES

A validation suite is implemented in a group of subclasses under a class called Tester. Each subclass corresponds to a tree class and has methods that exercise the corresponding tree method. Validation can be applied to an individual tree class or to the entire class hierarchy.

VALIDATE STRATEGIC ARGUMENTS

When a tree client sends a message that creates or modifies a tree, its arguments get validated. This protects against building incorrect trees and catches most errors at their point of origin. It provides a clearer explanation of the error than would be possible if it went undetected until later in the processing.

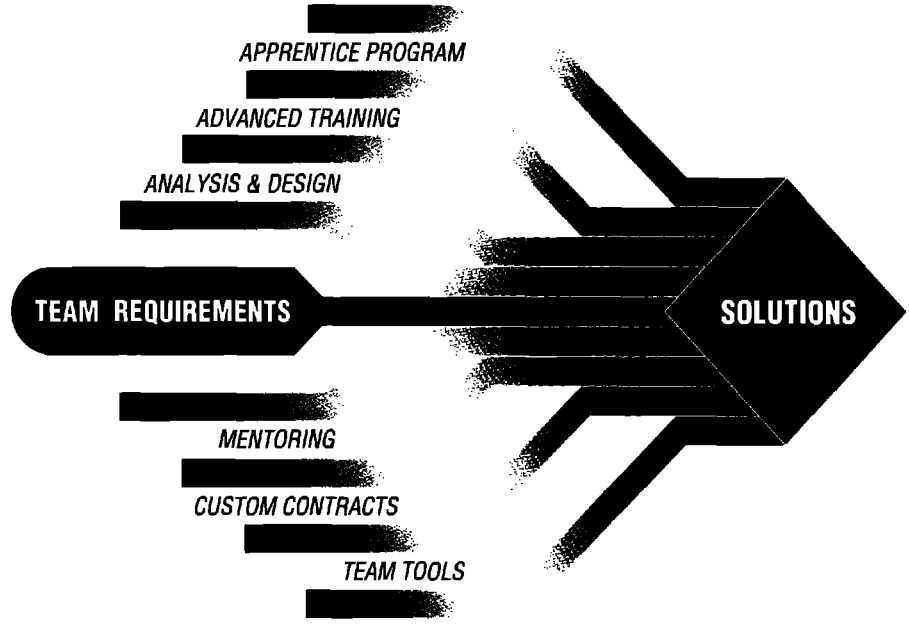
INTRODUCE SERVICES HIGH IN THE INHERITANCE HIERARCHY

The higher you place a method in an inheritance hierarchy, the more leverage you get. For example, once a collection class defines do:, it can use several enumeration methods defined by Collection. The tree classes work similarly. Most methods are implemented in Tree and do not need to be redefined by subclasses.

Three standard techniques are employed to gain leverage: indirect reference, semantic extension, and weak (or no-op) polymorphism. Examples include:

- ***Indirect reference.*** Subtrees are accessed via a message send rather than directly, allowing many variations of subtrees processing to be deployed.
- ***Semantic extension.*** A node's parent (superTree) is also accessed indirectly. The notion of root is extended to trees not containing a supertree pointer. They are defined to be their

Object Transition by Design



Object Technology Potential

Object Technology can provide a company with significant benefits:

- Quality Software
- Rapid Development
- Reusable Code
- Model Business Rules

But the transition is a process that must be designed for success.

Transition Solution

Since 1985, Knowledge Systems Corporation (KSC) has helped hundreds of companies such as AMS, First Union, Hewlett-Packard, IBM, Northern Telecom, Southern California Edison and Texas Instruments to successfully transition to Object Technology.

KSC Transition Services

KSC offers a complete training curriculum and expert consulting services. Our multi-step program is designed to allow a client to ultimately attain self-sufficiency and produce deliverable solutions. KSC accelerates group learning and development. The learning curve is measured in weeks rather than months. The process includes:

- Introductory to Advanced Programming in Smalltalk
- STAP™ (Smalltalk Apprentice Program) Project Focus at KSC
- OO Analysis and Design
- Mentoring: Process Support

KSC Development Environment

KSC provides an integrated application development environment consisting of "Best of Breed" third party tools and KSC value-added software. Together KSC tools and services empower development teams to build object-oriented applications for a client-server environment.

Design your Transition

Begin *your* successful "Object Transition by Design". For more information on KSC's products and services, call us at 919-481-4000 today . Ask for a FREE copy of KSC's informative management report: *Software Assets by Design.*

Knowledge Systems Corporation

OBJECT TRANSITION BY DESIGN

114 MacKenan Dr.
Cary, NC 27511
(919) 481-4000

own root, and all methods that manage the pointer work properly with them.

- ***Weak polymorphism.*** Trees use no-ops analogously to the 'close' method for streams in Smalltalk. A client can operate polymorphically without knowing what kind of tree or stream it is talking to.

INTRODUCE INSTANCE VARIABLES LOW IN THE INHERITANCE HIERARCHY

You should avoid introducing an instance variable until you need it. The flaw in the first two design iterations was to define instance variables directly in Tree rather than deferring the definition to subclasses. The third attempt takes this deferred approach. Instance variables are introduced one at a time as you descend the class hierarchy, and all possible combinations are available for anchoring application specific subclasses. The exception is that there are no classes defining a key without subTrees because it would make no sense.

The Smalltalk Collection classes again provide a good analogy. The superclasses of Array are Object, Collection, SequenceableCollection, and ArrayedCollection. As you descend this chain, capabilities for processing array elements are introduced. But an indexed instance variable to actually hold the data is not defined until the Array class itself. Even ArrayedCollection is dataless, allowing it to anchor several subclasses.

Third iteration shows number of methods defined by each class.

First attempt

```
Tree (key value subTrees)
    GroundedTree (superTree)
```

Second attempt

```
Tree (key subTrees)
    GroundedTree (superTree)
        FullTree (value)
    ValueTree (value)
```

Third attempt

```
Tree ()  100 methods
    PTree (superTree)     2 methods
        PVTree (value)    2 methods
    STree (subTrees) 26 methods
        SKTree (key) 40 methods
            SKPTree (superTree)  2 methods
                SKPVTree (value) 2 methods
            SKVTree (value)  2 methods
        SPTree (superTree)    2 methods
            SPVTree (value)   2 methods
        SVTree (value)    2 methods
    VTree (value)     2 methods
        BinaryTree (left right)     10 methods "defined for pedagogic purposes"
```

Naming conventions for third attempt

prefix derived from

```
K 'k' in key
P 'p' in superTree
S 's' in subTrees
V 'v' in value
```

Figure 1. Inheritance hierarchies in three design iterations.

INTRODUCE INSTANCE VARIABLES INTO THE INHERITANCE HIERARCHY IN THE RIGHT ORDER

One path of descent goes from Tree to STree to SKTree to SKPTree to SKPVTree. The variables are introduced in the order subTrees, key, superTree, and value. The ones requiring the most supporting methods (subTrees and key) are introduced first and the ones requiring the least supporting methods (superTree and value) are introduced last. SubTrees is introduced before key because a key cannot be defined apart from subTrees.

Figure 1 shows how many methods are defined by each class. Those with only two methods have basic read and write accessors for the instance variable they introduce. The hierarchy is arranged so that the two variables requiring the most support, subTrees and key, are introduced only in one class each. This prevented a major duplication of methods.

ACCESS INSTANCE VARIABLES INDIRECTLY

Kent Beck explains the advantages of accessing instance variables directly versus indirectly in the June 1993 issue of THE SMALLTALK REPORT. He favors direct access and uses indirect only when necessary. I'll try to show that for the tree classes, indirect access via a message send is indeed necessary. Let's consider encapsulation violation, availability of a variable, and interpretation of a variable.

Encapsulation violation is the main reason Kent gives for avoiding indirect reference. You don't want to publish a variable's read accessor unless clients legitimately need it. Otherwise, internal implementation details are exposed. Publishing the write accessor is even more serious. However, for the tree classes, clients need read access to all four variables and write access to three, so publishing them in a public interface is necessary and does not violate encapsulation.

Each instance variable is not available in each tree class. For example, several classes do not define or inherit the subTrees variable. Yet much of the processing done by a tree is performed on its subtrees. By using indirect access, it is possible to write general purpose methods at the top of the inheritance hierarchy that don't depend on the variable being present. If absent, a subtrees collection is synthesized from other data.

Some variables such as subTrees need to be interpreted flexibly. We might want to return its processed contents rather than raw contents. This can only be done with indirect reference.

INTERPRET INSTANCE VARIABLES FLEXIBLY

Using the subTrees variable again for a case study, we'll see how flexible interpretation contributes to the goal of reusability through subclassing. Three examples are static, dynamic, and synthetic interpretation.

Static interpretation is used with STree and some of its subclasses for branch nodes. They simply return the contents of their subTrees variable when queried.

Dynamic interpretation is used for an application specific subclass of STree (actually of SKPVTree) called roughly DirectoryTree. Rather than returning a static collection, it queries the file system dynamically for the contents of a directory, returns the

result in response to a subTrees query, and caches it in the subTrees variable for other purposes. This is an approximation of the actual implementation.

Synthetic interpretation is used for classes that do not have a subTrees variable. They synthesize a collection from other variables. For example, a BinaryTree class is defined for pedagogic purposes. Its left and right variables each stores a subtree. When queried for subTrees, a branch node returns an array containing the left and right nodes and a leaf node returns an empty array.

Synthetic interpretation is also used by STree, which has a subTrees variable, when answering queries to a leaf node. This is explained in the next section.

MAXIMIZE THE DATA CONTENT OF INSTANCE VARIABLES

An instance variable should hold useful information. Try to store something more informative than nil. The subTrees variable again provides an example.

For a branch node, subTrees holds a collection of subnodes. For a leaf node, it holds a collection class such as Array, OrderedCollection, SortedCollection, or Set. A leaf uses this class to know what kind of collection to instantiate if it later becomes a branch. Tree clients may specify the class when creating a node. Listing 1 provides sample code.

Other implementations are less effective. If a leaf stored nil in the subTrees variable, it would not have maximized the information content and would have denied clients the choice of collection class. If a leaf stored an empty collection instance in the variable, it would have taken more memory. It would also make it harder to distinguish between a branch with no subnodes and a leaf in a manner analogous to the way a file system distinguishes an empty directory from a file. If leaves and branches were implemented as distinct classes, the inheritance hierarchy would become unwieldy with leaf versus branch bifurcations. If a new variable were defined to hold the collection class for leaves, it would waste memory.

Listing 1: How STree interprets the subTrees variable.

```
STree: testing

isLeaf
    "Return a boolean indicating whether
    the receiver is a leaf node."

    ^self basicSubTrees isBehavior

STree: converting

makeBranch
    "Coerce the receiver to a branch node
    and return it."

    self isLeaf ifTrue: [self basicSubTrees: self basicSubTrees new]

STree: public accessing

subTrees
    "Return the subtrees, reporting an
    empty collection for a leaf node."

    ^self isLeaf
        ifTrue: [self basicSubTrees new]
        ifFalse: [self basicSubTrees]
```

DISTINGUISH PUBLIC AND PRIVATE METHODS

Although Smalltalk-80 does not enforce access restrictions to private methods, it at least allows them to be placed in a separate protocol. There are several reasons the tree classes make this distinction.

1. The superTree pointer is managed automatically. A public write method is not defined. Internal methods that need to set it use a private method.
2. A public read method provides processed access to the subTrees variable while the private method provides raw access.
3. Raw read and write access to all variables is needed in the tree copying machinery. It must be private.
4. Public write methods validate their arguments, while private write methods bypass validation for efficiency.

ASSESSMENT OF DESIGN PRINCIPLES

How well did these design principles contribute to the original goals? The intention was to create a hierarchy of Tree classes within the constraints of single inheritance that are robust, flexible, reusable, clear, simple, and reasonably efficient.

- ***Robust.*** The trees validate strategic arguments and offer a class validation suite.
- ***Flexible.*** The classes offer several measures of flexibility.
    1. All four instance variables are optional and are available in all legitimate combinations, enabling a client to store only the data needed for a particular problem.
    2. A client can access subtrees statically, dynamically, or synthetically.
    3. A client can choose from among several collection classes for storing subtrees.
    4. The various nodes within a single tree structure may use different collection classes for storing their subtrees.
    5. A branch node can store a full or empty collection of subtrees.
    6. It is possible to mix instances of different tree classes, such as STree and SKTree, in the same tree structure.
    7. It is possible to convert from one subtrees collection type to another or from one tree type to another.
- ***Reusable.*** There are several abstract tree classes. Most services are defined high in the hierarchy. All logical combinations of instance variables are available. They are accessed indirectly.
- ***Clear.*** The classes push the design principles to limits that might be unfamiliar to some programmers. To achieve clarity, I tried to write explanatory comments, use consistent naming conventions, and otherwise make consistent design choices.
- ***Simple.*** Ideally, each service should be implemented once. Single inheritance forced some services to be duplicated. This occurred in the nine classes labeled '2 methods' in Figure 1. Their read and write accessors resulted in 2 duplicate lines of code per class, or 18 total. This is acceptable.
- ***Efficient.*** Space is saved by defining instance variables low in the inheritance hierarchy and by storing maximal information in them. More savings come by common substructure sharing for applications not requiring a superTree pointer. Execution time is saved by bypassing argument validation whenever it is safe to do so. Although the cost for validation is modest, it can be bypassed entirely if necessary. A moderate penalty is incurred by implementing most of the methods with sufficient generality to place them at the top of the inheritance hierarchy.

CONCLUSION

I think the third design iteration satisfied these goals. Other programmers can confirm this by reusing the classes for their applications.

SOURCE CODE

Version 1.1 of the tree classes and their validation suites is available by anonymous ftp from the Smalltalk archives at the University of Illinois (st.cs.uiuc.edu 128.174.241.10) or the University of Manchester (mushroom.cs.man.ac.uk 130.88.13.70). Look in the directories for ParcPlace's VisualWorks or Objectworks. ■

Bruce Samuelson (bruce@utafll.uta.edu) uses ParcPlace Smalltalk for linguistic applications at the University of Texas at Arlington and with the Summer Institute of Linguistics.

INTERFACING SMALLTALK WITH REAL-TIME SYSTEMS: PART I

Edward Klimas

The development of complex real-time automation systems that must interact with client-server based factory information systems is one of the most promising applications of Smalltalk. Current applications of this technology not only include real-time commodity trading and financial transaction applications, but sophisticated instrumentation, mission critical biomedical, and process automation systems as well. The high productivity[1] and improved reliability[2] of this technology offer an impressive foundation for cost effectively deploying complex systems in the future.

Technically commendable real-time systems have been fielded in many of the currently available Smalltalk dialects.[3-6] One of the drawbacks to the wider application of this technology is the scarcity of documentation and example frameworks. The goal of this article is to explain a framework that can be used as a starting point for intermediate level Smalltalk developers to begin designing and developing Smalltalk-based real-time applications.

This article, the first of two on this topic, will follow the path of the messages from the low-level operating system application programming interface (API) calls up to the high-level Smalltalk event messages required to open, write to, and read from a device. A subsequent article will describe the message flow through the Smalltalk event mechanism as well as optimal multitasking issues.

This article will document a number of the steps that a typical real-time signal might take into and out of an OS/2-based Smalltalk/V-PM program. Due to its ready availability on many systems, the examples employed in this article are based on a serial port connection to an internal modem. Alternate sources for real-time signals can come just as easily from named pipes connected to external signals, real-time process I/O modules providing serially encoded ASCII data or from other coprocessor cards in the automation hierarchy.

LOW-LEVEL API CALLS

The basic path a real-world signal takes into the OS/2 system is typically dependent upon a specific device driver. Serial port signals, including those of internal modems, are routed through an OS/2 device driver called dos16DevIOCtl for 16-bit systems and dos32DevIOCtl for 32-bit applications. The architecture of this driver is based upon a curious singularity, in that the driver, for efficiency purposes, does *not* support interrupts to user programs. In an I/O-intensive application, the overhead of supporting an interrupt call can become a significant percentage of the processor's total computing requirements at maximum throughput rates. Hence, high speed communications are handled much more efficiently by filling I/O buffers and having programs asynchronously access the contents of those I/O buffers as soon as they are available. For support of multiple high-speed serial I/O streams, this approach is not only superior, but becomes imperative. An OS/2 serial port signal accesses Smalltalk via a call to the appropriate DevIOCtl API call. The calling conventions for the DevIOCtl API require the placement of parameters into registers. These parameters include a special parameter denoted by an ordinal number that identifies the specific system call being requested. When the registers are set, a software interrupt instruction is then issued transferring control to the OS/2 kernel. Using the ordinal number, the OS/2 kernel dispatches the system call to the appropriate routine. Upon completion, control returns to the requesting program with a return code that indicates the status of the requested operation.

Calls to APIs are typically quite straightforward, but they are also prone to excessive debugging time in any language because of the cryptic operating system error messages and frustrating system traps or halts that can result from incorrect API parameters. The following set of calls show debugged examples of working API calls into the DosDevIOCtl API. Upon inspecting the IBM documentation for the 32 bit OS/2 DosDevIOCtl system call in the on-line programmer's reference, one will find the C calling convention information to include:

```
/* DosDevIOCtl performs control
functions on a device specified
by an opened device handle. */
#define INCL_DOSPROCESS
#include <os2.h>
HFILE       DevHandle;
ULONG       ulCategory;
ULONG       ulFunction;
PVOID       pParmList;
ULONG       ulParmLengthMax;
PULONG      pParmLengthInOut;
PVOID       pDataArea;
ULONG       ulDataLengthMax;
PULONG      pDataLengthInOut;
APIRET      rc; /* Return code */

rc = DosDevIOCtl(DevHandle, ulCategory,
    ulFunction, pParmList,
    ulParmLengthMax,
    pParmLengthInOut, pDataArea,
    ulDataLengthMax, pDataLengthInOut);
```

The equivalent C-to-Smalltalk API call can be derived with the following "translation":

```
C       Smalltalk api: parameter
HFILE   handle
ULONG   ulong
PVOID   struct
PULONG  ulong
APIRET  ushort
APIRET  apiret    (also for /VOS2)
```

The basic procedure is to match up the OS/2 API call parameters with the corresponding Smalltalk api: parameters. The ordinal number for the corresponding IBM-defined API function, #284 for 32-bit DosDevIOCtl, is supplied in an ancillary file that comes with the Smalltalk/V-PM system. The equivalent Smalltalk/V version 2.0 for OS/2 code is:

```
DynamicLinkLibrary variableByteSubclass: #DosDLL
classVariableNames: 'OS2Errors '
poolDictionaries: ''

!DosDLL methods !
dos32DevIOCtl: aDeviceHandle
    deviceCatagory: aDeviceCatagory
    functionCode: aFunctionCode
    parameter: parameterListAddress
    parameterLengthMax: parameterLengthMax
    returnedParameterLengthInOut: returnedParameterLengthInOutb
    dataArea: dataAreaAddress
    dataLengthMax: dataLengthMax
            returnedDataLengthInOut: returnedDataLengthInOut
<api: '#284' handle ulong ulong
struct ulong ulong struct ulong ulong ushort >
^self invalidArgument
```

Unfortunately, the equivalent 16-bit call is different from the 32-bit call. As many developers may still need to use 16-bit APIs and Smalltalk/V-PM, the 16-bit equivalent is:

```
!DosDLL methods !
dos16DevIOCtl: data
    parameter: parameters
    functionCode: aFunctionCode
    deviceCatagory: aDeviceCatagory
    deviceHandle: aDeviceHandle
<api: '#53' struct struct ushort ushort short ushort >
^self invalidArgument
```

To make the resultant code fully portable across both 32- and 16-bit versions of Smalltalk, we implement our own generic devIOCtl: method that will call up the correct API, based upon the fact that VPMVMDLL fileName will return the version of Smalltalk/V being used as a string, (e.g., for Smalltalk/VPM 2.0, VPMVMDLL fileName = 'VPMVM20').

```
!DosDLL methods !
devIOCtl: data
    parameter: parameters
    functionCode: aFunctionCode
    deviceCatagory: aDeviceCatagory
    deviceHandle: aDeviceHandle
    "If this code is not running under VPM 2.0 then call a 16 bit api call
    to read/write to the I/O device, otherwise call the 32 bit API call"
    | returnedParameterLengthInOut returnedDataLengthInOut |

    "Get VPM version"
((VPMVMDLL fileName at: 6) asString asInteger >= 2)
        ifFalse:[ ^self dos16DevIOCtl: data
        parameter: parameters
        functionCode: aFunctionCode
        deviceCatagory: aDeviceCatagory
        deviceHandle: aDeviceHandle]

    ifTrue:[ ^self dos32DevIOCtl: aDeviceHandle
        deviceCatagory: aDeviceCatagory
        functionCode: aFunctionCode
        parameter: parameters
        parameterLengthMax: parameters size
        returnedParameterLengthInOut:
            returnedParameterLengthInOut
        dataArea: data
        dataLengthMax: data size
            returnedDataLengthInOut]
```

Table I. DosOpen/DosSet state flags.

PM	Constants Implemented	vHEX	Bit Pattern
OpenAccessReadonly	Y	0000	---- ---- ---- -000
OpenAccessWriteonly	Y	0001	---- ---- ---- -001
OpenAccessReadwrite	Y	0002	---- ---- ---- -010
OpenShareDenyReadwrite	Y	0010	---- ---- -001 ----
OpenShareDenywrite	Y	0020	---- ---- -010 ----
OpenShareDenyread	Y	0030	---- ---- -011 ----
OpenShareDenynone	Y	0040	---- ---- -100 ----
OpenFlagsNoinherit	Y	0080	---- ---- 1--- ----
OpenFlagsNolocality	N	0000	---- -000 ---- ----
OpenFlagsSequential	N	0100	---- -001 ---- ----
OpenFlagsRandom	N	0200	---- -010 ---- ----
OpenFlagsRandomSequential	N	0300	---- -011 ---- ----
OpenFlagsNocache	N	1000	---1 ---- ---- ----
OpenFlagsFailOnError	Y	2000	--1- ---- ---- ----
OpenFlagsWriteThrough	Y	4000	-1-- ---- ---- ----
20OpenFlagsDasd	Y	8000	1--- ---- ---- ----

For performance purposes, the VPMVMDLL version test might be determined at initialization time and cached in an instance variable, but in reality, this is not an expensive call under normal use.

Moving along the bottom up hierarchy, the calls to open the I/O device and read, write, and manipulate the resultant data are examined next.

The I/O channel is opened using a call to the DosLibrary class with the proper parameters in the call's mode. The ability to handle a broad spectrum of devices with little or no non-Smalltalk code is counterbalanced by myriad parameter selections. The most common parameters are implemented as PM-Constants that can be referenced directly from within Smalltalk itself. Table 1 conveniently lists all of the different modes avail-

able for the DosLibrary open: method, whether they are implemented as PMConstants, and the equivalent byte codes that can be used for those modes having no PMConstant.

```
ByteArray variableByteSubclass: #FileHandle
    classVariableNames: 'FileHandles '
    poolDictionaries: 'PMConstants '

!FileHandle class methods !
openComDevice: aString
"Answer a FileHandle for the device named aString."
of | aHandle result anActionCode |
    aHandle := self new: 2.
    anActionCode := ByteArray new: 2.
    result := DosLibrary open: aString asParameter
        handle: aHandle
        action: anActionCode
        initSize: 0
        attribute: 0
        flags: 1
        mode:
            "report errors via return code"
            OpenFlagsFailOnError|
        "OpenFlagsNocache-
        I/O is not cached-(no need to reread info)"
        16r1000|
        "file handle is private to current process and may
        not be inherited by child processes"
        OpenFlagsNoinherit|
        "permit read write sharing"
        OpenShareDenynone|
        "read/write access"
        OpenAccessReadwrite
    reserved: 0.
    (result = 0)
        ifFalse: [^DosDLL OS2Error: result].
    ^aHandle
```

An unsuccessful attempt to open the device will contain a non-zero error code as the result value from the DosLibrary open: message. The OS2Error: method will help in explaining what the problem was.

Once the I/O device has been successfully opened, we read the device as follows:

```
!SerialPort class methods !
read
"To catch serial input during this read operation, read the contents of
the serial port receive buffer recursively until there is nothing in the
buffer"
| result aString qSize aLagniappe bytes |
qSize := self numRecQChar.
(qSize < 1) ifTrue:[^nil].
    aString := String new: qSize.


"Depending upon 16 or 32 bit versions use the appropriate call to read input"
"Get VPM version"
((VPMVMDLL fileName at: 6) asString asInteger >= 2)
    ifFalse:[ self comPortHandle "16 bit VPM"
        readInto: aString
        atPosition: 1.]
    ifTrue:[ result := DosLibrary
        read: self comPortHandle "32 bit VPM"
        buffer: aString
        bufSize: aString size
        bytesRead: (bytes := PMStructure new: 2) asParameter.
        (result = 0)
            ifFalse: [ ^DosDLL OS2Error: result ]].

"Check if any more data came in while we were reading the buffers and
recursively read the data in. Normally this should never occur, but just
in case."
aLagniappe := self read.
aLagniappe isNil ifTrue: [^aString].
^(aString , aLagniappe)
```

THE SUPPORTING FRAMEWORK

The effective interaction with a real-world signal requires a supporting infrastructure of commands to set and get the parameters for input and output (I/O). The following methods show several examples of opening, reading, and writing values to the I/O device. Although these examples provide the supporting framework for serial I/O, most other devices will use simple permutations on most of the same types of program calls.

```
Object subclass: #SerialPort
    instanceVariableNames: ''
    classVariableNames: 'ComPortHandle '
    poolDictionaries: 'CharacterConstants '

!SerialPort class methods !
baud
"Return baud rate"
| result aDataArea |
"The dataArea contains or receives the data"
aDataArea := ByteArray with: 0 with: 0.
result := DosLibrary devIOCtl: aDataArea
    parameter: nil
    functionCode: 16r61 "query baud rate"
    deviceCatagory: 1
    deviceHandle: self comPortHandle.
(result = 0)
    ifFalse: [^DosDLL OS2Error: result ].
^(aDataArea asPMLong lowHalf)

baud: anInteger
"Set baud rate"
| result aParameter |
"Create a word parameter with the baud rate in it"
aParameter := (PMStructure new: 2)
            shortAtOffset: 0
            put: anInteger.
result := DosLibrary devIOCtl: nil
        parameter: aParameter asParameter
        functionCode: 16r41 "set baud rate"
        deviceCatagory: 1
        deviceHandle: self comPortHandle.
(result = 0)
    ifFalse: [^DosDLL OS2Error: result ]

closeComPortHandle
    self comPortHandle close.
    ComPortHandle := nil
```

```
comPortHandle
"Use lazy initialization to set up the comport handle the first time the
handle is required"
(ComPortHandle isNil)
    ifTrue: [ComPortHandle :=
        (FileHandle openComDevice: 'COM2')].
^ComPortHandle

comPortHandle: aString
^ComPortHandle := FileHandle openComDevice: aString

dataBits
"get the number of data bits
    5   5 data bits
    6   6 data bits
    7   7 data bits (initial value)
    8   8 data bits"
^((self lineCharacteristics) at: 1)

dataBit: anInteger
"set the number of data bits
    5   5 data bits
    6   6 data bits
    7   7 data bits (initial value)
    8   8 data bits"
| aLineCharacteristic |
aLineCharacteristic
    := self lineCharacteristics.
aLineCharacteristic
    at: 1
    put: anInteger.
self lineCharacteristics:    aLineCharacteristic

initializeComPort
"Initialize COM port for tests"
    self baud: 2400.
    self dataBit: 8.
    self stopBits: 0.
    self parity: 0.

numRecQChar
"Return the number of characters in the receive Q"
^(self receiveQ asPMLong lowHalf)

numTranQChar
"Return the number of characters in the transmit Q"
^(self transmitQ asPMLong lowHalf)

lineCharacteristics
"Set line characteristics"
| result lineCharacteristics |
lineCharacteristics := ByteArray
                with: 0
                with: 0
                with: 0.
result := DosLibrary
    devIOCtl: lineCharacteristics
    parameter: nil
     "query characteristics"
    functionCode: 16r62
    deviceCatagory: 1
    deviceHandle: self comPortHandle.
(result = 0)
    ifFalse: [^DosDLL OS2Error: result ].
^lineCharacteristics

lineCharacteristics: aLineCharacteristic
"Set line characteristics for COM2"
| aDeviceHandle result |
result := DosLibrary
    devIOCtl: nil
    parameter: aLineCharacteristic
     "set line characteristics"
    functionCode: 16r42
    deviceCatagory: 1
    deviceHandle: self comPortHandle.
(result = 0)
    ifFalse: [^DosDLL OS2Error: result ]

receiveQ
"Return info about the number of characters
in the receive Q and its size"
| result aDataArea |
aDataArea := ByteArray
        with: 0 with: 0 with: 0 with: 0.
result := DosLibrary
        devIOCtl: aDataArea
        parameter: nil
"get the number of receive queue chars"
        functionCode: 16r68
        deviceCatagory: 1
        deviceHandle: self comPortHandle.
(result = 0)
    ifFalse: [^DosDLL OS2Error: result ].
^aDataArea

parity
"get the line parity
    0 no parity
    1 odd parity
    2 even parity (initial value)
    3 mark parity (parity bit always 1)
    4 space parity (parity bit always 0)"
^((self lineCharacteristics) at: 2)

parity: aParity
"set the line parity
    0 no parity
    1 odd parity
    2 even parity (initial value)
    3 mark parity (parity bit always 1)
    4 space parity (parity bit always 0)"
| aLineCharacteristic |
aLineCharacteristic :=
        self lineCharacteristics.
aLineCharacteristic at: 2 put: aParity.
self lineCharacteristics:    aLineCharacteristic
```

The combination of all of these methods should be integrated into a test method for debugging purposes and regression testing of changes, as well as for documenting the appropriate use of the framework.[7] An example follows:

```
!SerialPort class methods !
selfTest
"Test script for debugging interface to the DosDevIOCtl API"
    "SerialPort selfTest"
| inspectData |
```

```
CursorManager execute change.
"Initialize COM port for tests"
self initializeComPort.

"Test valid baud rates"
#(110 150 300 600 1200 1800 2000 2400 3600 4800 7200 9600 19200 )
    do:[ :aValidBaudRate |
        self baud: aValidBaudRate.
        (self baud = aValidBaudRate)
            ifFalse:[self error:
                'Baud rate set/get failure']].

"Test valid data bits"
#(5 6 7 8)
    do:[ :aValidDataBit |
        self dataBit: aValidDataBit.
        (self dataBits = aValidDataBit)
            ifFalse: [self error:
                'Data bit set/get failure']].

"Test valid stop bits
Note: 1.5 stop bits only valid for 5 bit word length
     2 stop bits not valid with 5 bit word length "
#(0 2)
    do:[ :aValidStopBit |
        self stopBits: aValidStopBit.
        (self stopBits = aValidStopBit)
            ifFalse: [self error:
                'Stop bit set/get failure']].

"Test parity bits"
#(0 1 2 3 4)
    do:[ :aValidParityBit |
        self parity: aValidParityBit.
        (self parity = aValidParityBit)
            ifFalse: [self error:
                'Parity bit set/get failure']].

self sizeOfTransmitQ.
self numTranQChar.

"Reset the com port to receive some characters"
self initializeComPort.

"Send a string to test a modem on COM2"
self comPortHandle deviceWrite: 'ATI3\' withRealCrs.

"Wait 3/4 of a second for the modem to test & respond"
DosLibrary realSleep: 750.
inspectData := self read.

"Be sure to close the Com port handle or a subsequent
selfTest will fail to open it successfully"
self closeComPortHandle.

"open up an inspector on the data. This line should be commented out
once the framework has been debugged"
inspectData isNil ifFalse: [inspectData inspect].

CursorManager normal change.
"return a true value if we reach this point with no errors, otherwise
walkbacks should have appeared"
^true
```

An extension to class String for adding a carriage return to a string is provided for sending messages to the COM port:

```
FixedSizeCollection variableByteSubclass: #String
    classVariableNames: ''
    poolDictionaries: 'CharacterConstants'

!String methods !
withRealCrs
"Answer the receiver string where each occurrence of the character \
has been replaced with a carriage return character."
    1 to: self size do: [ :index |
        (self at: index) = $\
            ifTrue: [self at: index put: Cr]]
```

Depending upon the device that is being communicated to, a linefeed may also be required.

CONCLUSION

The expressive power of Smalltalk and its class libraries can be used to deal with real-time data just as easily, simply and effectively as with graphical user interfaces and database information.

The next article in this series will deal with the issues associated with moving the real-time data efficiently through the Smalltalk event system. ■

Acknowledgments

The support of Digitalk's Michael Chin and Kris Severson is gratefully acknowledged.

References

1. Harmon, P. Texas Instruments chooses O-O technology for a CIM project, OBJECT-ORIENTED TECHNOLOGIES, 2 (10): 1–13, 1992.
2. Dotts et. al. Experience report—development of reusable test equipment software using Smalltalk and C, Addendum to OOPSLA 92 Proceedings.
3. Barry, B. Real-time object-oriented programming systems, AMERICAN PROGRAMMER, October, 1991.
4. Duimovich et al. Smalltalk and Embedded Systems, DR. DOBB'S JOURNAL, October 1991.
5. Dehli et al. STEAMEX: a real-time expert system for energy optimization, AIENG 89-APPLICATIONS OF ARTIFICIAL INTELLIGENCE IN ENGINEERING, Cambrige U.K. 11–14 July1989.
6. Klimas, E. Quality assurance issues for Smalltalk-based applications, THE SMALLTALK REPORT 1(9): 3–7, 1992.

Ed Klimas is Managing Director of Linea Engineering Inc., a supplier of custom object oriented solutions for automation and industrial applications. Ed, along with coauthors Dave Thomas and Suzanne Skublics of OTI, is writing a book on developing commercial Smalltalk-based systems titled SMALLTALK WITH STYLE. He can be reached at 216.381.8493.

SMALLTALK IDIOMS

Kent Beck

Helper methods avoid unwanted inheritance

THE TOPIC OF this issue's column on Smalltalk idioms, following the general theme of inheritance, is how to manage the use of super. Several issues back I wrote a column entitled "The Dreaded Super" in which I catalogued all the legitimate (and otherwise) uses of super in the existing Smalltalk/V and VisualWorks images. I'm still very proud of that column, but a couple of days ago I discovered I had left out one very important technique in dealing with super.

The pattern that follows, Helper Methods Avoid Unwanted Inheritance (not my best name ever), tells how to resolve this problem. The first time I remember anyone talking about the problem was when Richard Peskin of Rutgers brought it up on the net several years ago. A lively "discussion" ensued. The solution is one many Smalltalkers have discovered over the years.

Before I jump into the pattern itself, let me say a word about patterns in general. *Hot.* That's the word. Grady Booch and Dick Gabriel have both been trumpeting patterns in other SIGS publications. Ralph Johnson has had a couple of ECOOP/OOPSLA papers published on them. Pete Coad has jumped on the bandwagon in his OOP book (although I think he's missing the point). I have gotten a half dozen calls in the last month or so from people who have heard about my interest and want to tell me what they are doing with patterns.

I think *patterns* will be the next big buzzword in the object world. If you want to get involved, now is a great time to try writing some patterns of your own. Don't get discouraged if your first efforts don't sparkle. It took me six years to get my first pattern that I didn't want to immediately crumple up and throw away. It shouldn't take you nearly as long.

Here are some criteria I use when evaluating a pattern:

- Does it make me change my program? The best patterns don't just say, "Hey, here is a useful configuration of objects." The patterns I find most powerful say, "If you find yourself with this problem, create this useful configuration of objects and it will be solved."
- Does it explain its assumptions? Each pattern implicitly contains assumptions about what is most important about the decision it describes. If a pattern says, "We want simple programs, we want fast programs, we want programs we can write quickly, but in this case the most important thing is getting the program running quickly," I have a much better basis for evaluating it.
- Does it contain an illustration? Good patterns can invariably be reduced to a single picture. Drawing that picture, or writing a code fragment example can sharpen your understanding considerably.

Give it a try. I'd be glad to critique your efforts, or you could try passing them around to other Smalltalk or C++ programmers you know.

PATTERN: HELPER METHODS AVOID UNWANTED INHERITANCE

Context

When you are using "super" at the bottom of a three-deep inheritance tree, you may find yourself wanting to inherit the root class's behavior, but not the immediate superclass's.

Problem

In this case, you almost want to be able to say something stronger than super, like "give me that class's method but no one else's." Experience with C++, which has such a facility, says that using such a feature is a maintenance nightmare. How can you take advantage of inheritance, share code, and remain within Smalltalk's simple control structures?

Constraints

- ***Code sharing.*** The resulting program should contain as much code sharing as possible.
- ***Use inheritance.*** The resulting code should use inheritance. Inheritance may be important for simplifying the implementation of the rest of the class.
- ***Simple code.*** The result should be no more complex than necessary. This recommends against using Delegation or some other pattern that requires extensive code changes.

Solution

Put the behavior you don't want to inherit in its own method. Invoke that method from the method that contains the send to "super". Override the new method in the subclass to either do nothing, or replace its behavior with behavior appropriate to the subclass (Figure 1).

Example

This problem often occurs in initialization code. *continued on p. 20...*

GETTING REAL

Juanita Ewing

Abstract classes

This month, I will discuss abstract classes and talk about why they are really useful. Abstract classes are classes that don't have any instances and can be grouped into two categories: implementation-based and design-based. Implementation-based abstract classes often have many methods that are inherited and used directly by subclasses. A design-based abstract class may not have any methods that can be used directly by subclasses.

The example code in this article is from version 2.0 of Smalltalk/V for Macintosh. Classes have been simplified for the purpose of illustration.

ABSTRACT VS. CONCRETE CLASSES

Concrete classes usually have both behavior and state. Point is a concrete class. It has state, two instance variables x and y, and it has behavior, such as + and -.

Design-based abstract classes provide a specification for subclasses. This is like having a detailed on-line design document. A design-based abstract class usually has behavior, but not state. That is, there are no instance variables, and there is no indexable part defined by the class. Abstract classes are most useful when they are as general as possible. Any state provided by an abstract classes limits subclasses, since they inherit specification of the state. Design-based abstract classes are often part of a framework.

Implementation-based abstract classes are generally based on the behavior of existing classes and are created afterwards. Common methods are identified in two classes, and moved to a new superclass. The new superclass is generally an implementation-based abstract class. It contains many concrete methods, formerly duplicated in the subclasses. An implementation-based abstract class may have state that corresponds to its behavior. The motivation for creating an implementation-based abstract classes is to locate common code in one place, which is therefore easier to maintain.

DESIGN-BASED ABSTRACT CLASS EXAMPLE

The well-known Smalltalk class Magnitude is the focus of this section and can be found in every Smalltalk image. Magnitude, a design-based abstract class, has no state. Magnitude is part of the informal framework of objects in the Smalltalk image: it supports requests for ordering. For example, the default collaboration between SortedCollection and its elements requires elements to respond to the <= message.

Magnitude is a generic class that could be specialized for many applications. The pitch of musical notes and the latitude and longitude components of map coordinates are possible domain specific subclasses of Magnitude. Date, Time, and Number are subclasses of Magnitude in the Smalltalk class library. Because it is an abstract class, there are never instances of Magnitude in a Smalltalk system, but there are instances of its concrete subclasses Date and Time.

Let's examine some of the details of Magnitude that make it a good example of a design-based abstract class. The comment for Magnitude indicates its purpose:

The class Magnitude is an abstract class defining behavior common to all objects for which an ordering is defined.

The role of an design-based abstract class is to provide a specification for subclasses. Subclasses of Magnitude must provide an implementation of the methods that Magnitude specifies

```
<
<=
>
>=
between:and:
comparableWith:
max:
min:
```

The implementation of a method can be inherited. Magnitude provides default implementations for all methods except <. If you examine the class (see Listing 1), you will notice that all the other methods are defined in terms of <, = and their derivatives. All subclasses of Magnitude inherit a default implementation of = from Magnitude's superclass Object.

The default implementations make it easier to extend hierarchies because developers only need to implement a small number of methods. Subclasses can inherit the rest of the methods. In the case of Magnitude, a developer would only have to implement <. It is likely that developers would also want to override = to complete the ordering algorithm for their subclass. (If you override = then you must also override hash to match, or you will get errors when you attempt to use containers, such as Set and Dictionary, that are based on hash.)

Several key characteristics make Magnitude a good example of a design-based abstract class:

- Magnitude has the right amount of behavior: not too much to understand but enough to perform a useful set of functions.
- Magnitude's behavior is cohesive. All methods are related to ordering. Its role is clear, which makes it easier to use and understand Magnitude.
- New subclasses are much easier to add to the hierarchy if the required methods are specified by a superclass using a special designator such as implemented BySubclass. The alternative is to require a developer to deduce the required set of methods.
- Magnitude has no state, which avoids implementation limitations on subclasses.
- Magnitude provides default implementations, which make it easier to develop new subclasses. Developers need only write a minimal number of methods.

Most design-based abstract classes arise because a developer has made a conscious effort to create an abstract class that fills a role. Design methodologies, such as responsibility-driven design, detail how to translate a specification into a design, which are implemented with both abstract and concrete classes. An interesting example is the evolution of the MVC framework, which is discussed in the article, "Reimplementing Model-View-Controller" by Leibs and Rubin (Smalltalk Report, Volume 1 Number 6).

DISCOVERING ABSTRACT CLASSES

Have you ever developed two classes with the same set of public messages, but different implementations? If so, then you have a situation that would benefit from an implementation-based abstract class. A good rule of thumb is to make an implementation-based abstract superclass whenever you have two hierarchically unrelated classes with common behavior. Implementation-based abstract classes are typically derived from existing classes rather than designed from scratch.

Follow these steps to make a common superclass for the two classes:

1. Create a new superclass.
2. Rearrange the hierarchy so the subclasses inherit from the new superclass.
3. Move all identical methods from the subclasses to the common superclass. Even if the two classes have radically different implementations, there are usually some methods in common.
4. For each remaining selector that the two classes have in common, create a method with that selector in the common superclass. The body of the method should be a designation for the subclasses to implement the method. In Smalltalk/V, the body of the method is usually self implementedBySubclass. In Objectworks, the body of the method is usually self subclassResponsibility.

The role of the common superclass, an implementation-based abstract class, is to contain implementations that are appropriate for all subclasses. For ease of evolution and maintenance, each piece of functionality should be in exactly one place. If code is duplicated, it is likely to be fixed or modified in only one place, introducing new errors and inconsistencies. During active development, a new subclass is much easier to add to the hierarchy if a known set of methods must be implemented, rather than requiring a potentially new developer to deduce the public set of methods.

Listing 1. The implementation of some of Magnitude's instance methods.

```
< aMagnitude
    Answer <true> if the receiver is less than <aMagnitude>."
    ^self implementedBySubclass
<= aMagnitude
    "Answer <true> if the receiver is less than or equal to<aMagnitude>."
    ^self < aMagnitude or: [self = aMagnitude]
> aMagnitude
    "Answer <true> if the receiver is greater than <aMagnitude>."
    ^(self <= aMagnitude) not
```

CONCLUSION

Using our rule of thumb, developers can create new superclasses whenever they see unrelated classes with a common set of public methods. Often, these new common superclasses are implementation-based abstract classes.

Design-based abstract classes are a powerful mechanism for creating extensible hierarchies. They provide an on-line specification and default implementations. Well-designed abstract classes allow developers to create new subclasses with minimal effort.

In my next column, I will show you how to use abstract classes to write platform-independent code. ■

Juanita Ewing is a senior staff member of Digitalk Professional Services, 921 SW Washington, Suite 312, Portland, OR 97205, 503.242.0725.

THE BEST OF comp.lang.smalltalk

Alan Knight

Extending the environment: part 2

Last month's column described a number of improvements to the Smalltalk environment, all of them incremental changes that improved the existing environment. This month, we discuss some packages that take a more radical approach, completely overhauling the basic tools.

I'll limit myself to discussing three packages, all freely available by ftp. I've avoided commercial packages, although many of them make significant changes to the development tools, since they're better dealt with in reviews or product news. All three are for ParcPlace Smalltalk (a.k.a. ObjectWorks or VisualWorks) since it has more packages freely available than Smalltalk/V. Finally, I've arbitrarily chosen three that sounded promising. There are certainly others, and if you know of one that might be of interest, please let me know about it. I'd be interested, for example, in looking at extended Smalltalk/V environments.

Development environments are a religious issue, and I'm sure most readers will disagree with my ideas. I'll try to be more informative than judgemental, but please bear in mind that much of what I'll say is subjective. This is particularly true of the sections on weaknesses, which contains many things that aren't necessarily problems. You should also bear in mind that my impressions of all these packages are based on limited use. I would have preferred to spend a couple of days doing real work in each package, but didn't quite dare. My work at the time required use of ENVY/Developer and I didn't want to risk work that other people were paying for with my limited knowledge of the ENVY internals. Finally, these packages are evolving and there may be new and improved versions available by the time you read this.

Enough excuses. If you want to try these for yourself, all are available from the standard ftp servers at st.cs.uiuc.edu (University of Illinois) and mushroom.cs.man.ac.uk. (University of Manchester). I'll list directories for the Illinois server. The corresponding directories can be found on the Manchester server under /pub/goodies/uiuc.

CLASSBROWSERS

This package was written by Carl Watts (carl@parcplace.com). Carl works at ParcPlace, but this is an independent project of his, not a ParcPlace product. On the UIUC server it's available in the directory /pub/st80_vw/ClassBrowsers.st. There is no README or help file, but I've extracted some remarks from the class comments:

> **The most important features of the ClassBrowser is that it always shows you inherited attributes as well as attributes locally defined in the class. Attributes (like representation variables and methods) that are locally defined are shown in bold.**
>
> **...The menus are more context-sensitive than many other Smalltalk applications. The menus are constructed to be sensitive to whatever you have selected. The menu items are very different if you have something selected than if you don't.**
>
> **The metaphor for moving something (like a method) to a different place (like a different protocol) is to select the item you want to move, select 'take it...' from the menu, select the place you want to move it to (like the protocol), and then select 'and move it here...' from the menu of the destination.**

This package is the least disorienting of the three for someone accustomed to the standard browsers. It simplifies the system by reducing the number of browsers to two, a class browser and a system organization browser. The system organization browser has only two panes: one showing categories and one showing the classes in the selected category. From here it's possible to open a class browser, similar to the standard class browser, but with additional features. It shows inherited methods and variables at all times, with a button to control the visibility of Object methods. It also has a list of instance/class variables that can be manipulated through the menus.

Notable Features

- Allows browsing inherited methods and variables
- Reduces the number of different browsers
- Changes the menus to be more consistent and more context-sensitive
- Good extended navigation tools, such as "self senders," "self messages," and "browse overrides"
- Very nice install/uninstall feature. Once the code has been filed in to the image, a one line "doit" switches between the normal and extended browsers.
- The concept of a globally selected object. In most places that an object can be selected, the "take it..." menu item is available. Once taken, the object can be used as an argument for many other operations. This is very nice for binary

operations such as moving classes between categories, rearranging inheritance hierarchies, or changing the protocols of methods. It's far better than the standard method of prompting for the name of another needed object.

- Allows manipulation of the representation through menus. Rather than editing the text representation of the class, menus can be used to add, remove, or rename instance variables.

Weaknesses

- Some of the features that sounded most interesting were not yet implemented in the version I used. That may have changed by the time this is printed.
- I like to have more control than this allows over browsing inherited features. The class browser does show inherited methods, but you can't find out what class they are actually implemented in and the only control over which inherited methods are shown is the toggle for Object methods. There is also no hierarchy browser, which I found awkward. Nevertheless, while my initial reaction is to consider these problems, they could also be considered features. They force you to consider a class only in terms of what it defines and what it inherits, ignoring the implementation detail of where inherited features originate. I might like it once I get used to it.
- There's not enough overlap in features between the two browsers. I found myself jumping back and forth between them to do operations where I didn't think it should be necessary.
- Since there's a pane listing instance/class variables, I'd like to see a feature similar to the Digitalk class browser, where selecting a variable in that pane can be used to limit visibility to methods accessing that variable.

ISYSE

This was written by **Deeptendu Majumder** (dips@cad.gatech.edu) and is available in the directory pub/st80_r41/ISYSE. The help file describes its objectives:

> **I developed this as a tool to enhance the programming environment for my research work, and many of the features are there because I felt the need for them. You may or may not have a similar working style and needs, and hence the usefulness of this tool will vary. The high-level requirements that drove this effort include:**
>
> - **Reduce the need for opening new windows. I want to open windows only when needed.**
> - **Introduce graphical information access capabilities. (I am a great believer in graphical information access)**
> - **Provide some amount of history management.**
> - **Provide some support for saving and restoring ongoing changes. Let the user switch work contexts when looking for information and go back to a previous context when ready.**
> - **Allow users to follow a thought process in their tasks.**

In contrast to the ClassBrowsers, this system is radically different from the normal Smalltalk environment, and quite intimidating at first glance. There are seven panes in the main browser, including a bright green graphical view of the class hierarchy across the top (Figure 1). This minimizes the number of windows required at the cost of greatly increasing the size and complexity of the windows used. Don't expect to use this on a small screen.

I wasn't able to get completely comfortable with this environment in the limited time I had to play with it. My overall impression was that of a tool written by someone for their own

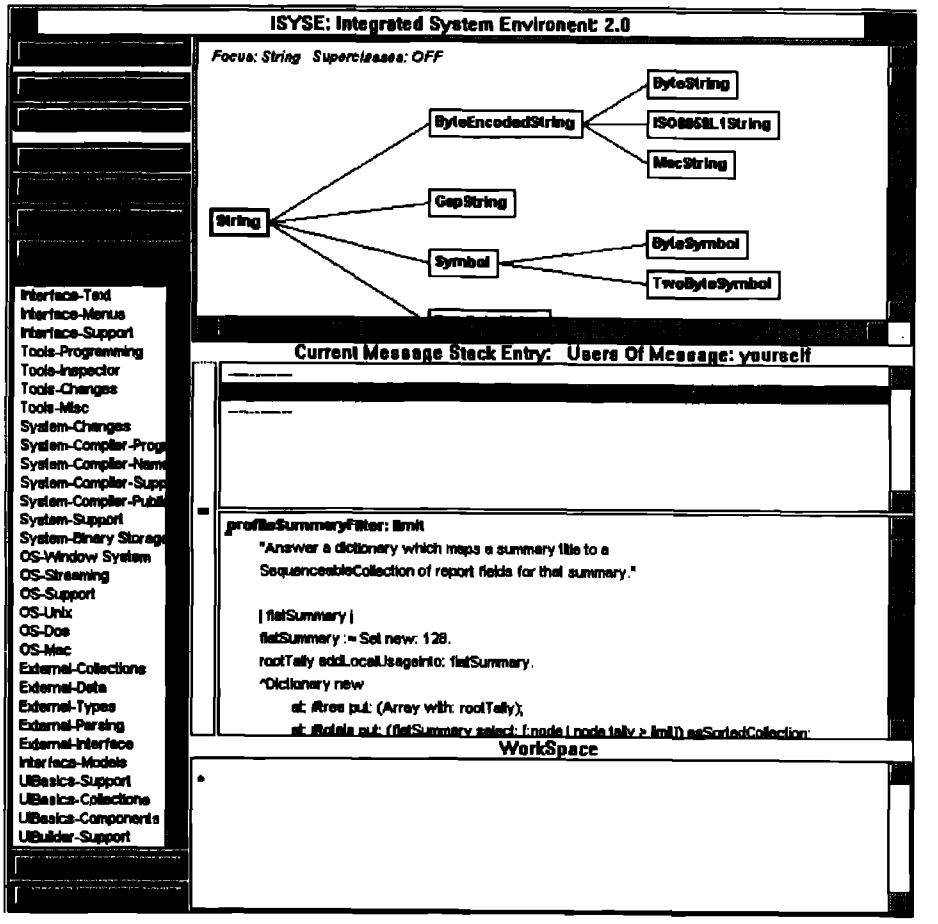


Figure 1. One of the ISYSE browsers.

use that isn't too polished yet. To the author it's entirely natural, but it can be very confusing for others. I've written enough of those tools to know the feeling. This is, however, very powerful, and might well be worth the effort of learning or customizing it to your own intuition. It builds on the work of several smaller "goodies" to provide a wealth of features.

Notable Features

Apart from the features described eariler, I also noted

- On-line help
- Sliders to resize text views dynamically
- Use of graphical views to present information
- Easy switching of work contexts
- Addition of many advanced features
- More use of buttons (as opposed to pop-up menus)

Weaknesses

- Robustness. The system is not as polished as the other two I examined. It was written on a SPARC, and I had some problems getting it to run under MS-Windows (filenames, hard-coded X font names, and a few mysterious system crashes).
- Speed of graphics. The graphical view of the class hierarchy is quite nice, but very slow for large hierarchies. The first thing I figured out how to do was switch it to the hierarchy for Object, and it took me several minutes after doing that to bring the system back to a usable state, since every screen update required a long wait.
- Confusion. The system is very confusing at first glance, and has some unusual concepts. The purpose of the browser buttons isn't immediately clear, and when the system starts up many of them do nothing. Once you understand the system, this makes sense, since these buttons are part of the history mechanism, and there's no history yet. Until you understand the system it's very annoying.
- Similarly, the menu mechanism in the graphical view can be disorienting. I expected to select one of the visible classes and then operate on it. Instead, there is a pop-up menu that changes depending on where the cursor is at that moment. It's more like the way Smalltalk behaves across views than within a view.

NEWTOOL

NewTool was written by Dan Walkowski (walkowsk@cs.uiuc.edu) and is available in /pub/st80_r41/NewTool. He writes:

> **NewTool is an alternative Browser. It looks substantially different, has more of the features at the top level, and has several nice characteristics:**
>
> - **All NewTool windows update instantly and automatically to reflect the current state of the system, whether the change was made from a CodeTool, a debugger, or a filein.**
> - **Screen real estate is used more efficiently. Windows are tiled automatically, and instead of each window wasting a third of its area as a navigation mechanism, there is only one that controls all windows.**

Why did you write this thing, Dan?

> **On a recent visit to UBILAB in Zurich, I was given a demo of Sniff, a C++ browser/debugger/editor by its original author, Walter Bischifoberger. It is extremely well done, with lots of well-thought-out features.**
>
> **If you use NewTool a bit, you'll realize that I tried to reduce the six or so different Browsers in the original image down to only one CodeTool with lots of features. This saves screen real estate, and as long as I don't try to cram too much into one window, I think it's faster and easier to use, too.**

NewTool looks quite different from the standard browsers, but is pretty easy to pick up. It creates a "Navigator" down the left hand side of the screen, which is similar to the top four panes of the system browser. This navigator controls all of the CodeTools, except the ones that are frozen. Having half a dozen browsers on the screen all showing the same code is not very useful, so most browsers are normally frozen. A CodeTool in which code is being edited is automatically frozen. Also, from a frozen CodeTool you can change the navigator to reflect that CodeTool's information, effectively switching contexts. This is a nice concept, elegantly implemented, and one that definitely relieves some of the screen congestion that can be caused by browsers with many panes (Figure 2).

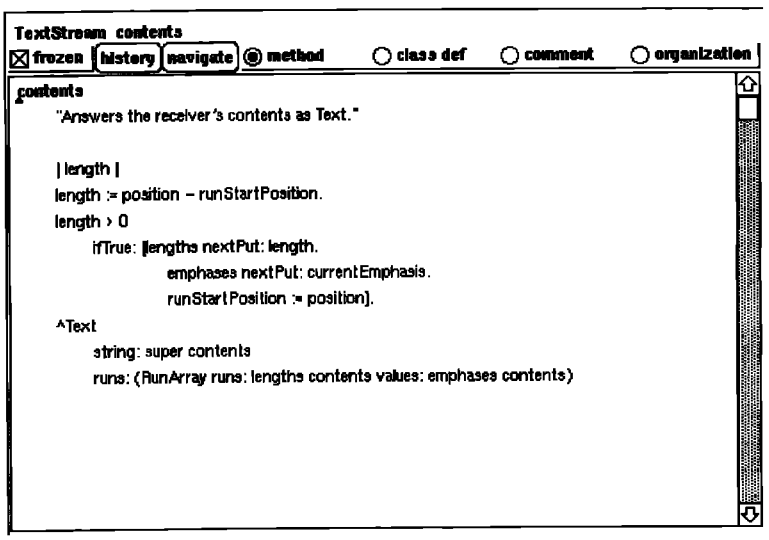


Figure 2. A CodeTool (from the NewTool package).

...continued from page 14

Notable Features

- Auto-update of windows (a feature I've always wanted)
- Separating the navigation and text-editing windows
- The concept of "freezing" a text-editing view. This allows you to have one navigation tool, but to still have browsers open on several different classes/methods and to navigate among them fairly easily.

Weaknesses

My biggest problem with NewTool is that, while addressing some issues very nicely it leaves many others open. For example, it doesn't deal with browsing inherited methods. Addressing some of these issues without making the windows too large and complicated is the challenge.

CONCLUSIONS

All three of these systems significantly improve some aspect of the standard browsers. All have their weaknesses. You've probably skipped to the end wondering which to try out, or which I'd choose to use in my own work.

To recommend a single best choice is difficult. All have interesting concepts and nice features, and although they all extend the same environment, they mostly attempt to solve different problems. Which you want to try depends on which problems you consider most important.

For me, the answer is simple. I plan to take the pieces I like from all three and incorporate them in my own personal environment along with anything else that strikes my fancy. Two other ideas I'd like to make use of are ENVY/Developer's use of multi-selection and SmalltalkAgents support for saving extended text attributes with method text.

Finally, I'd like to commend all of these people for having the courage and the generosity to make their code available for public use and criticism. I'll, close with an inspirational quote from **Dan Walkowski:**

> **I think I have made a significant improvement on the original Browser. What do you think? Every Smalltalk programmer I know has some gripes with the Browser, and yet so few do anything about them. I'm throwing down the gauntlet with NewTool. Let's have a contest of sorts. If you think NewTool isn't quite right, or isn't even close to your ideal browser, fix it! Let's combine all of our best ideas and develop the best programming tools around. Then we all win.** ■

Alan Knight works for The Object People, 509-885 Meadowlands Dr., Ottawa, Ontario, K2C 3N2. He can be reached at 613.225.8812 or as alan_knight@mindlink.bc.ca.

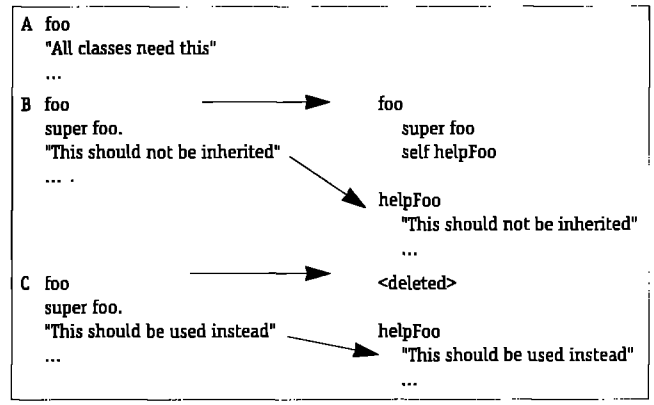


Figure 1. Invoking helper methods avoid unwanted inheritance.

```
A>>initialize
    "Initialize generic structure"
    ...

B>>initialize
    super initialize.
    "Allocate machine resources"
    ...
```

Now we want to write C, a subclass of B that uses other machine resources. If we write:

```
C>>initialize
    super initialize.
    "Allocate other machine resources"
    ...
```

We will allocate B's machine resources and C's, too, which is not what we wanted. If we don't send to super, we don't get the generic initialization from A. If we copy A's initialization into C, we have a multiple update problem with the code in A and C. The right solution is to use Compose Methods to introduce a helper method in B that allocates machine resources:

```
B>>initialize
    super initialize.
    self initializeMachineResources

B>>initializeMachineResources
    "Allocate machine resources"
    ...
```

Then we can override initializeMachineResources in C:

```
C>>initializeMachineResources
    "Allocate other machine resources"
    ...
```

We can delete initialize in C. The logic in B works just fine to invoke its specialized behavior.

This solution satisfies all of the constraints: Inheritance is still used, the maximum amount of code is being shared, and the resulting code is only slightly more complex than the original.

You may have to invoke Compose Methods before you can separate out the method you want to override but not invoke. ■

Kent Beck is 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, 408.338.4649 (v), 408.338.3666 (f), or 70761,1216 on CompuServe.

 ...continued from page 1

perclass. PMI then creates methods in the new class which forward messages to the PMI-superclass instance. A method is added to create this instance, which may be edited by the programmer to accommodate special instance-creation needs. All new methods are placed in a new method protocol, named to identify it as implementing PMI. The VisualWorks forms for browsing and editing PMI relationships are sensitive to these names, and use them to determine what PMI relationships a class has defined.

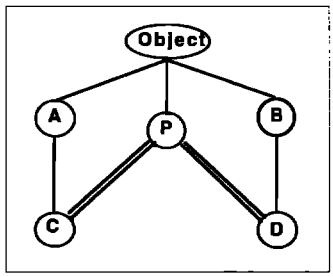


Figure 1. Simple PMI example.

For example, in the desired class structure shown in Figure 1, classes C and D both need to inherit from class P, but the class hierarchy doesn't easily allow classes A and B to each be subclasses of P. PMI creates instance variables in C and D to refer to (different) instances of P, and creates appropriate methods in C and D to forward relevant messages to their respective instances of P.

A pair of VisualWorks forms are used to allow browsing existing classes for previously defined PMI relationships, and to create new or delete existing PMI relationships. One form allows selection of the class to be manipulated (or browsed) by a pattern match on class names. The other uses a fixed target class, passed in to the form when it is created.

To create a new PMI relationship, a class is chosen to inherit from. This class and all its superclasses (including Object) are presented in a selection list. The programmer can select any or all of these classes, and the set of all instance methods defined by these classes is presented in another selection list. The programmer chooses the methods to be included with the definition, and clicks a Generate button. This causes the instance variable, the new protocol, and all selected methods to be created in the target class. Note that only instance methods are presented and defined here. The interface allows the user to define more methods for a previously existing PMI relationship (in case you don't get it right the first time ;-).

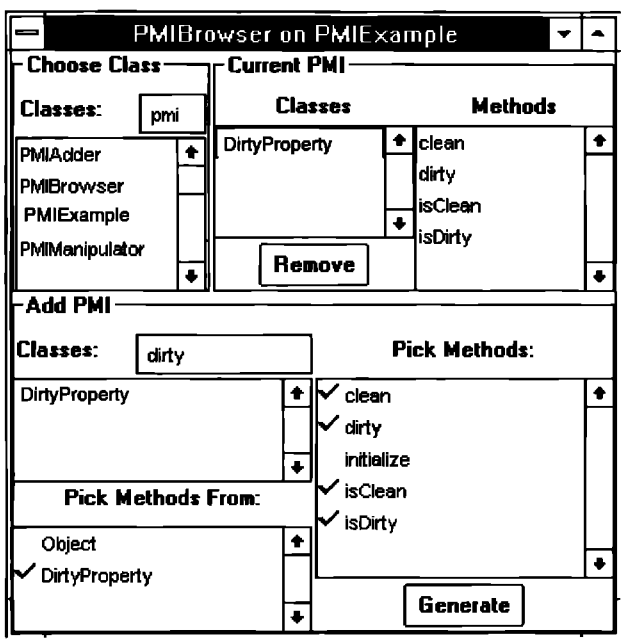


Figure 2. Example of adding dirty property.

To browse existing PMI relationships, a target class is chosen (depending on the VisualWorks form, as noted above). The VisualWorks form determines existing PMI relationships and methods from the protocols implemented by this class, and presents existing PMI superclasses in a selection list. Choosing one of these classes causes a list of the methods defined specifically for that class to be presented in another selection list. Indirection methods can be deleted individually (via a pop-up menu), or the PMI superclass may be removed completely (all methods, the protocol, and the instance variable) by clicking a Remove button.

KINDS OF PROPERTIES

Two kinds of properties are supported by PMI: properties that are orthogonal to the subclass and those that are more intimate with the containing instance. An *orthogonal* property represents some state that is completely independent of the containing object. The dirtiness property discussed above is a good example. The state of this property, and all manipulations of it, are independent of any state of the containing instance; an object is dirty or not—it doesn't depend on other state of the object.

An *embedded* property in some way knows about the object that contains it, and its state manipulation is dependent on this

knowledge. For example, if an embedded property would normally use methods defined in its subclasses, it must be able to invoke these in the PMI-subclass. To support embedded properties, PMI supplies class EmbeddedProperty, with a single instance variable, embedding, intended to refer to the containing object. All subclasses of EmbeddedProperty are recognized by PMI when creating a PMI inheritance, and the embedded instance is automatically initialized to refer to the containing object. Methods in an EmbeddedProperty may use the construct self embedding to refer to the containing object (i.e., an EmbeddedProperty must be coded knowing it may be used this way). For convenience, if the embedding isn't initialized, self embedding answers self, so an EmbeddedProperty may be inherited from and coded to work independent of whether its actually embedded.

EXAMPLE

To illustrate the use of PMI, this section presents a DirtyProperty class, and an example of how it might be used. The definition of the simple class DirtyProperty is (edited for brevity):

Class definition:

```
Object subclass: #DirtyProperty
    instanceVariableNames: 'dirty '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Pragmatic-Multiple-Inheritance'
```

Instance Methods:

```
clean
    "Assert the object is clean."
    dirty := false

dirty
    "Assert the object is dirty."
    dirty := true

isClean
    "Is the object is clean?"
    ^dirty not

isDirty
    "Is the object is dirty?"
    ^dirty

initialize
    "Assert the object is clean to start with."
    self clean
```

Class Methods:

```
new
    ^super new initialize
```

An example of the PMI browser being used to add the DirtyProperty to a class is shown in Figure 2. In this example, the class PMIExample has added DirtyProperty as a PMI superclass. The methods clean, dirty, isClean, and isDirty have been defined in PMIExample, and these will be forwarded to an instance of DirtyProperty, which is created on demand. In this manner, a PMIExample instance will respond to the DirtyProperty protocol as if it had directly inherited from DirtyProperty. To further illustrate how PMI is implemented, the resulting PMIExample class code is presented below:

```
Object subclass: #PMIExample
    instanceVariableNames: 'aDirtyPropertyForPMI '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Pragmatic-Multiple-Inheritance'

aDirtyPropertyForPMI
    ^aDirtyPropertyForPMI == nil
        ifTrue: [aDirtyPropertyForPMI := self
createaDirtyPropertyForPMI]
        ifFalse: [aDirtyPropertyForPMI]

clean
    ^self aDirtyPropertyForPMI clean

createaDirtyPropertyForPMI
    "Edit this method to customize creation of the DirtyProperty
instance."
    ^DirtyProperty new

dirty
    ^self aDirtyPropertyForPMI dirty

isClean
    ^self aDirtyPropertyForPMI isClean

isDirty
    ^self aDirtyPropertyForPMI isDirty
```

The aDirtyPropertyForPMI instance variable and all the presented instance methods were generated by the PMI browser, in protocol PMI:DirtyProperty. For subclasses of EmbeddedProperty, the createaXxx method also initializes the embedding to be the containing object.

WHAT PMI DOESN'T DO

PMI does not implement the semantics of full multiple inheritance. For example, the virtual base class concept in C++ isn't supported (this allows sharing of a single base class instance when multiple superclasses each have the same base class). There is also no support for the PMI-subclass to directly override methods in a PMI-superclass. An EmbeddedProperty object may be used in some cases to provide the desired semantics, albeit using a different coding technique (self embedding). Class methods are not supported in the current version of PMI. It is not yet clear that they are needed, but they should be easy to add (instead of an instance variable, the class can be referenced directly).

RESULTS AND FUTURE IDEAS

We have applied PMI to an application in which 5 classes (out of approximately 120) were removed by mixing in the DirtyProperty. This included several classes removed by collapsing the class hierarchy, since the need for an abstract dirty superclass was removed. The need for EmbeddedProperty was apparent in trying to refactor some user interface classes needed for database logon. There were three cases of database usage (two variants of middleware to access Oracle relational data, and one to access Servio's GemStone object data). Prior to PMI, a DBMS accessing abstract superclass had been artificially added to the class hierarchy to enable this, which provided the function but not in a form reusable by possibly different components. Moving the DBMS logon functions to a separate EmbeddedProperty subclass allowed the class hierarchy to reflect the actual desired structure,

by mixing-in the DBMS logon function where needed. The DBMS logon class relies upon the subclass to provide other context information (e.g., which actual dialog class to use), and thus needs to access the containing object via self embedding. PMI thus enabled the removal of all Gee-multiple-inheritance-would-be-nice-here comments in this application.

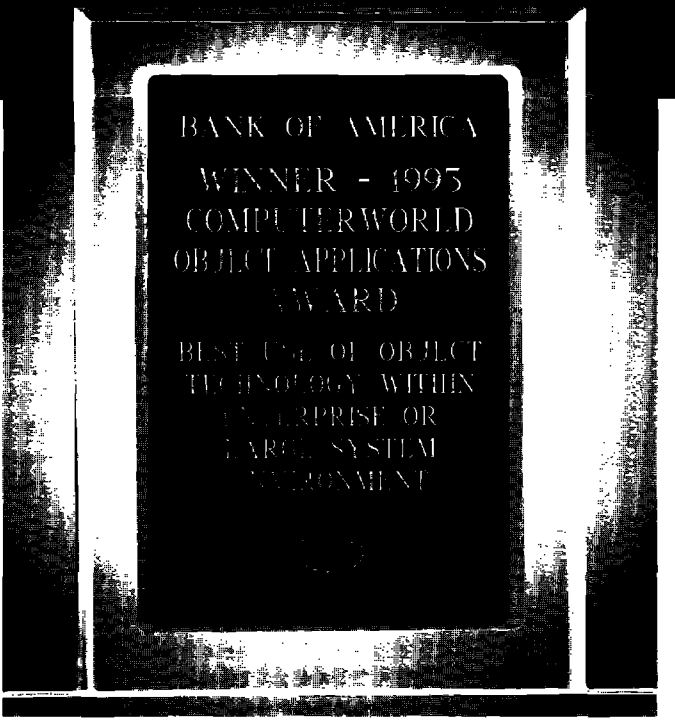
Additional features that might be added to PMI include: adding support for class methods, integration with the system browser (perhaps via a spawn-PMI menu item), and providing additional information for classes to be used as PMI-super-classes (e.g., to more readily identify the methods most likely to be useful to inherit, for generating forwarding methods). It could be argued, however, that the class comment should document the appropriate methods (since this is part of the class specification), but an automated mechanism would be helpful.

SUMMARY

PMI supports a useful subset of multiple-inheritance semantics in Smalltalk without damaging its intuitive single-inheritance model. It also supports a notion of mixing-in properties, as defined by other classes, which seems to be the useful, intuitive, and understandable subset of multiple-inheritance semantics. PMI is supported by two VisualWorks forms that make it easy to create, browse, and remove PMI relationships. ■

Bob Beck is a Principal Engineer in the Object Technology group at Sequent Computer Systems, Inc. He may be reached at rbk@sequent.com or Sequent Computer Systems, Inc., 15450 SW Koll Parkway, Beaverton, OR 97006.

LOOK WHAT HAPPENED WHEN DIGITALK BROKE INTO THE BANK.



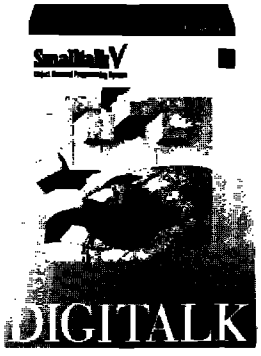
Congratulations to Bank of America on their new 11-state wide area network. A system they call "the most sophisticated distributed network in the world."

With good reason. Their network configuration tools have already won the Computerworld 1993 Award for Best Use of Object-Oriented Technology within an Enterprise or Large System Environment.

Of course, that's what happens when a company like Bank of America turns to a powerful technology like Digitalk's Smalltalk/V.

LIKE MONEY IN THE BANK.

Why are so many Fortune 500 companies like B of A switching to Smalltalk/V? Smalltalk/V lets you show prototypes of enterprise-wide systems in weeks instead of months. In fact, systems as ambitious as Bank of America's can be completed in as little as 18 months.



In addition, our Team/V Group Development Tool lets large teams of programmers use version control to easily coordinate their work. Plus you'll be surprised at how quickly your in-house staff becomes productive with Smalltalk/V.

The bottom line is Smalltalk/V helps a company get more done in less time. Which can save very large amounts of corporate cash.

RATED #1 BY USERS TOO.

On behalf of Computerworld, Steve Jobs presented the award to Bank of America. But industry luminaries and Fortune 500 managers aren't the only ones who have recognized the value of Smalltalk/V. Users have discovered that Smalltalk/V is the only object-oriented technology that's 100% pure objects. With hundreds of reusable classes of objects, thousands of methods and 80 object classes specifically designed to build GUIs fast. Which means no more time spent writing code from scratch.

BANK ON SMALLTALK/V.

So it's no wonder that so many companies are doing award-winning work with Smalltalk/V. Incidentally, Smalltalk/V applications can be easily ported between Windows, OS/2 and Macintosh. And you can distribute 100% royalty-free.

For information on how Digitalk's Smalltalk/V can save you time and money, call 1-800-531-2344 department 310 for our special White Paper. And be sure to ask about Digitalk's Consulting and Training Services.

Call right now, and see how Smalltalk/V can yield a maximum return on your investment.

SMALLTALK/V. 100% PURE OBJECTS. DIGITALK