THE Smalltalk REPORT

SIGS PUBLICATIONS

Publishers of JOURNAL OF OBJECT-ORIENTED PROGRAMMING, OBJECT MAGAZINE, C++ REPORT, THE SMALLTALK REPORT, THE X JOURNAL, REPORT ON OBJECT ANALYSIS & DESIGN, OBJECTS IN EUROPE, and OBJEKT SPEKTRUM (GERMANY)

Cover illustration by Dave Cutler/SIS

Table of Contents

June 1995 Vol 4 No 8

Features

Columns

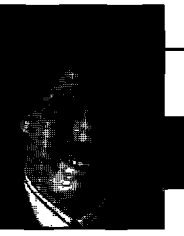
Departments

The Smalltalk Report (ISSN# 1056-7976) is published 9 times a year, monthly except in Mar–Apr, July–Aug, and Nov–Dec. Published by SIGS Publications Inc., 71 West 23rd St., 3rd Floor, New York, NY 10010. Second Class Postage Pending at NY, NY and additional Mailing offices. Canada Post International Publications Mail Product Sales Agreement No. 290386.

Individual Subscription rates 1 year (9 issues): domestic $89; Mexico and Canada $114, Foreign $129; Institutional/Library rates: domestic $199, Canada & Mexico $224, Foreign $239. To submit articles, please send electronic files on disk to the Editors at 885 Meadowlands Drive #509, Ottawa, Ontario K2C 3N2, Canada, or via Internet to streport@objectpeople.on.ca. Preferred formats for figures are Mac or DOS EPS, TIF, or GIF formats. Always send a paper copy of your manuscript, including camera-ready copies of your figures (laser output is fine).

POSTMASTER: Send domestic address changes and subscription orders to: The Smalltalk Report, P.O. Box 5050, Brentwood, TN 37024-5050. For service on current domestic subscriptions call 1.800.361.1279 or fax 615.370.4845. Email: subscriptions@sigs.com. For foreign subscription orders and inquiries phone +44(0)1858.435302. PRINTED IN THE UNITED STATES.

Editors' Corner



John Pugh

Paul White

One of the points we always emphasize when we talk to clients embarking on their first Smalltalk project is that the problems they will face will be as much cultural as technical. Unfortunately, reference materials dealing with the cultural issues are very difficult to find. There is a real need for informative articles dealing with process, requirements tracing, testing, reward systems, etc. To help fill this void, we are very pleased to welcome our new co-columnists Jan Steinman and Barbara Yates. Jan and Barbara have been involved in numerous large Smalltalk projects and have a great deal of "front line" experience to share with us through their "Managing Objects" column.

In what we think is a very positive step for the Smalltalk industry, 11 of the leading companies have joined together to form the Smalltalk Industry Council (STIC); STIC is a non-profit trade association dedicated to growing the Smalltalk market. The initial members of STIC are: American Management Systems, Easel Corporation, IBM Corporation, Knowledge Systems Corp., Linea Engineering, Objectshare Systems Inc., Object Technology International Inc., ParcPlace Systems Inc., RothWell International Inc., Servio Corporation, and The Object People. There will undoubtedly be many more members by the time you read this.

STIC was formed with the following mandate:

- Establish Smalltalk as the object-oriented environment of choice for corporate developers.
- Create a focal point for the Smalltalk community.
- Listen and respond to the needs of Smalltalk users.
- Encourage the participation of all segments of the Smalltalk industry.
- Encourage standards for Smalltalk.

This mandate is very similar to ours as editors of The Smalltalk Report and so we enthusiastically endorse the efforts of STIC and its executive director, Reed Phillips. If STIC is to be truly representative of the Smalltalk community, however, it must have strong participation from the user community. Users are noticeably absent from the current list of members (as is at least one major Smalltalk vendor!). STIC can provide a unified voice for the Smalltalk community when needed but it should also be an important focal point for Smalltalk users to voice their concerns. For membership information contact STIC at the address below.

As its first project, STIC commissioned International Data Corporation (IDC) to study the market perception of Smalltalk in relation to other procedural and object-oriented programming languages. The study, entitled "Smalltalk Market Accelerates," concluded that: Smalltalk is more compatible with typical corporate developer skills than competing object-oriented languages; Smalltalk is gaining popularity in corporate MIS; and misconceptions about Smalltalk (in areas such as speed, memory requirements, use of garbage collection, and steep learning curve) are outdated. Based on a telephone survey of 296 corporate developers with typically 15 years programming experience, the study makes excellent reading for anyone embroiled in a language decision debate or wanting to get an appreciation of how Smalltalk is being perceived and used in the corporate world.

The study paints a very rosy picture for the future of Smalltalk stating that Smalltalk is the fastest growing O-O language (vendor revenues are estimated to rise from $56 million in 1994 to over $250 million by 1998) and that many organizations are already developing large mission-critical systems with Smalltalk. As documented in the study, the strengths of Smalltalk for enterprise-wide development are many and far too numerous to list here. The list of perceived weaknesses is much smaller. Here are three of the main ones: (1) the lack of experienced Smalltalk programmers; (2) the need for better deployment options (e.g., smaller runtime images) and better mechanisms (binary format) for distribution of and use of third-party class libraries; and (3) the need for better interoperability with networks, GUIs, and databases. No big surprises here!

For STIC membership information or a copy of IDC report #9818, "Smalltalk Market Accelerates," contact the Smalltalk Industry Council at 919/821-0181, info@STIC.pdial.interpath.net.

We hope you enjoy this issue!

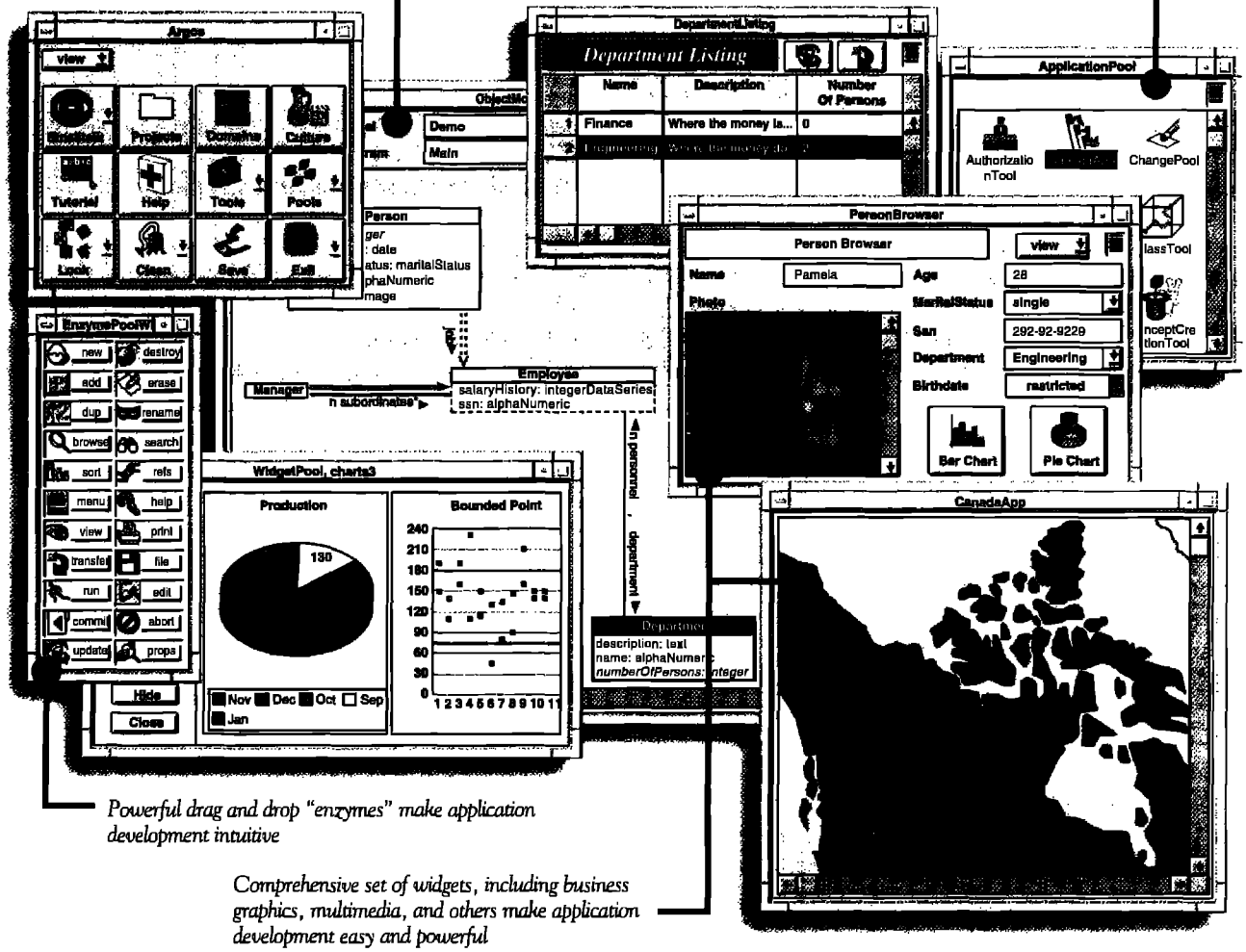
Editors' Note added in proof: *Just as we were going to press we heard of the announcement that two major Smalltalk vendors, ParcPlace and Digitalk, have signed an agreement to merge the two companies. The new company will be named ParcPlace-Digitalk, Inc. The merger, expected to be completed before the end of August, is subject to the approval of the companies' shareholders. According to a joint press release the merger is not expected to effect the delivery of either company's next product release. ParcPlace expects to release the next version of VisualWorks in the fourth quarter of 1995. Digitalk expects to deliver a release of Visual Smalltalk Enterprise for Windows 95 and Windows NT/Server in the fourth quarter of 1995. Long-term product plans are expected to be announced at the ParcPlace International Users Conference, July 20–Aug 2, 1995, in San Jose, CA.*

Introducing Argos

The only end-to-end object development and deployment solution

An integrated object modeling tool provides model-driven development for enterprise-wide applications

All object models are managed in a shared repository, supporting team development and traceability



Powerful drag and drop "enzymes" make application development intuitive

Comprehensive set of widgets, including business graphics, multimedia, and others make application development easy and powerful

VERSANT Argos™ is the only application development environment (ADE) that makes it easy to build and deploy powerful, enterprise-wide object applications. Easy because Argos features an embedded modeling tool and Smalltalk code generation that ensure synchronization between your models and applications. Powerful because Argos supports full traceability and workgroup development through a shared repository.

Argos automatically generates multi-user database applications that run on the industry-leading VERSANT ODBMS. Argos deals with critical issues such as locking and concurrency control transparently. And only Argos is packaged as a completely visual ADE built on ParcPlace VisualWorks®.

Leading organizations — in industries from telecommunications to finance — are using Argos to deliver business-critical applications. Find out how Argos can help you deliver your critical applications in weeks, instead of years.

Contact us today at 1-800-VERSANT, ext. 415 or via e-mail at info@versant.com

VERSANT
The Database For Objects™

1380 Willow Road • Menlo Park, CA 94025 • (415) 329-7500

Segregating application and domain

Tim Howard

THIS IS THE SECOND ARTICLE IN A SERIES of three dedicated to the topic of segregating application information and domain information in VisualWorks application development. The first article presented the case of why it is essential that an application have a strict segregation between its application information and its domain information (THE SMALLTALK REPORT 4[8]). This second article discusses the implementation of *domain objects*, the keepers of the domain information. The third article will cover the application classes that provide the user interface for the domain objects.

In the first article, we talked about the need to bundle domain information into cohesive objects relevant to the problem space. A typical example would be an address object that references four pieces of domain information—the street, city, state, and zip. More than likely, these four pieces of information would be strings. The first article also used the example of an employee object that contains all kinds of information relevant to an employee of a company. Such information probably includes various strings, numbers, and Booleans. Because all employees have an address, it is quite likely that an employee object also contains an address object. Objects such as employee and address are referred to as *domain objects*. A domain object is a logical container of purely domain information, usually represents a logical entity in the problem domain space, and is void of any dependents or model behavior. The remainder of this article covers the general characteristics of domain objects and introduces the class DomainObject—an abstract superclass for all domain objects. Source code for DomainObject, as well as examples, is available from the archives at the University of Illinois (st.cs.uiuc.edu).

GENERAL CHARACTERISTICS

The DomainObject class has been created to provide the common characteristics of all domain objects. Its superclass is Object but this could be changed for the benefit of a particular persistent object store. All domain information in an application should reside in objects that are a kind of DomainObject.

Some may refer to a domain object as a domain *model*. This is true to the extent that these objects are modeling the domain problem space. For example, all the domain objects making up an airline reservation system may be referred to as the "airline reservation domain model." However, the term "model," as used in Smalltalk, strongly implies an object that has dependents, and, as was pointed out in the first article, domain objects should not have dependents. Therefore I prefer the term domain object to the term domain model.

It is very important that domain objects avoid model-type behavior at all cost. They should not deal in any way with interface issues and they should have no dependents. In general, domain objects should not:

- contain models such as ValueHolders or SelectionInLists
- have dependents,
- deal with user interface issues
- contain non-domain information
- perform application type functions

Domain objects should know how to do certain things, however. In general, they should know how to:

- copy themselves completely and correctly
- compare themselves to other domain objects of the same type
- provide testing and other services concerning their domain information
- facilitate other objects that choose to print or display them

As a developer, you should minimize the amount of direct communication you have with a domain object. Such communication should be restricted to the application model, and ultimately the user, as much as possible. Remember, the domain object exists as a way to bundle certain domain information logically into a single, cohesive object and to model the domain space. Sometimes, however, it is necessary to directly manipulate the domain object. In such cases, it is your responsibility as the developer to ensure that any visual updates are initiated because there is no dependency mechanism to do it for you!

AGGREGATION AND ASSOCIATION

When designing domain object classes, it is very important to keep track of which references indicate aggregation and which references indicate association. Often times a domain object references other domain objects because it is *made up of*, or *composed of*, these other domain objects. This concept of composition is referred to as an *aggregation*. An aggregation is a reference to an object that specifies composition. An employee object, for example, could be an aggregation of an address object, a work history object, a name object, and perhaps other objects as well. That is, an employee object is composed of these other objects and each instance of Employee has its own unique instance of Address, WorkHistory, etc.

THE OBJECT PEOPLE

PRESENT...

TOPLink

A Smalltalk/Relational Database Interface

TOPLink allows Smalltalk applications to make full use of relational database facilities in an efficient manner with a minimal impact on your application. TOPLink is designed to allow objects to be mapped to relational databases using mechanisms which are independent of the Smalltalk model designed by your team, allowing your application to take advantage of the power of Smalltalk and the performance and robustness of relational databases.

TOPLink *provides a full, object-level persistence mechanism, that supports all of these features and more, including the following:*

- objects can be stored across multiple tables;
- multiple objects can be stored in a single table *(i.e., each row in a table can contain one or more objects);*
- full support of object identity and caching;
- multiple sessions and multiple database systems;
- support for inheritance;
- support for stored procedures; and
- full proxy support for complex object instantiation.

TOPLink **is extensible and provides you with the ability to tune your application to maximize the performance you require.**

TOPLink *is currently available for IBM Smalltalk, VisualWorks and Visual Smalltalk.*

TOPLink *supports many database systems, including Sybase, Oracle, DB2/2, Informix, Paradox, dBase, Btrieve, and others.*

What's so great about TOPLink?

Lots of people sell relational database interfaces for Smalltalk...

The difference between TOPLink and many of the relational database interfaces available is its ability to store and retrieve objects, and not just row data. Many interfaces will allow you to associate a class with a table, and then copy the data from a row in that table to a new instance of that class. Unfortunately, that is not sufficient.

To be useful, it should be possible to store and retrieve the relationships between objects as well as the actual data that makes up the object. You should be able to handle data types that the database does not support *(such as symbols);* objects that contain references to themselves *(either directly or indirectly);* objects that have references to other, complex objects; and many other features that are ***fully supported by*** TOPLink.

TOPLink...

the next generation relational database interface

The Object People Inc.

509-885 Meadowlands Dr.
Ottawa, Ontario, K2C 3N2
Phone: (613) 225-8812 FAX: (613) 225-5943

109 Upper Shirley Avenue
Southampton, England S015 5NL
Phone: 44 1703 775566 FAX: 44 1703 775525

E-mail: info@objectpeople.on.ca

There are times, however, when one domain object maintains a reference to another domain object for the purpose of illustrating a relationship. In such cases, the perception is that the first domain object *knows about* the second but does not *contain* it, per se, as part of its aggregation. This is what is referred to as an *association*. An association is a reference to an object that does not indicate composition. For example, an employee object may reference a company object as its employer and reference a second employee object as its supervisor. We would never consider an employee to be composed of a company nor composed of other employees, yet it is necessary to maintain these references for purposes of indicating the supervisor and employer relationships. When an employee object changes supervisors, it merely breaks the reference to the employee object that is currently associated as its supervisor and references a new employee object instead.

The one-to-many relationship is a fiction, however, and you should recognize it as such.

Unfortunately, there is no real distinction between an aggregation reference and an association reference in Smalltalk because they are implemented the same way—that is, with instance variables—and it is only a matter of perspective that draws the distinction. When one object references another, it is not always apparent whether the first object is composed of the second object or just trying to illustrate an association with the second object. Therefore, it is up to the designer of the domain object, and the developer who uses it, to know when an instance variable is used to indicate an aggregation relationship or an association relationship. This distinction will have profound effects on domain objects with respect to copying, persistent storage, and initialization, as well as other matters.

DOMAIN OBJECT STRUCTURE

For convenience, the objects referenced by a domain object can be divided into four groups: atomic objects, mutable objects, collections of atomic objects, and collections of mutable objects. Each of these presents its own set of unique problems for managing the domain information. The first two groups express a one-to-one relationship between the domain object and the object it is referencing. The last two groups express a one-to-many relationship between the domain object and the elements in a collection.

In most applications, there are certain types of objects that should not be edited directly but instead should be replaced with another object of a similar type. Such objects are referred to as *atomic objects*. Atomic objects are those perceived to be the smallest units of information from which the problem space can be described, and they should not be edited but, instead, replaced by another atomic object. If you look at a domain object as a tree structure, then the atomic objects are the leaf nodes of that tree. Usually counted among the atomic object types are the standard literal data types such as strings, integers, floats, dates, and Booleans. Atomic objects can also be other domain objects that are referenced by association. Such domain objects, as well as strings, are examples of atomic objects that *can* be edited but *should not*, in order to maintain the integrity of their atomic nature. It is the developer's responsibility to ensure that a domain object's atomic information is never edited, only replaced.

Mutable objects are objects referenced by the domain object that can be edited directly. Such objects are other domain objects referenced as part of the original domain object's aggregation. As was illustrated previously, an employee object might contain a work history object, address object, and name object—each being another type of domain object and part of the employee object's aggregation. It is necessary to ensure that the instance variables that reference other domain objects are properly initialized—either in the accessor method, as is shown below, or in an initialize method.

```
address
    "Return the employee's address."

    ^address isNil
        ifTrue: [address := Address new]
        ifFalse: [address]
```

A domain object can contain a collection of other objects. In this way, domain objects can express a one-to-many relationship with other objects. These collections, for the most part, are loosely typed. That is, all the elements in the collection are usually of the same type or a similar type. The collection's accessing method should initialize the collection if necessary (or it should already have been done in an initialization method). Suppose we have a Company domain object that maintains a collection of vendors in its vendors instance variable. The vendors accessing method might look like the following.

```
vendors

    ^vendors isNil
    ifTrue: [vendors := OrderedCollection new]
    ifFalse: [vendors]
```

Be careful when expressing one-to-many relationships. A designer might say that one student object references many course objects. This one-to-many relationship is a fiction, however, and you should recognize it as such. The reality is that a student object references a collection object that references many course objects. The actual implementation of a one-to-many relationship in Smalltalk introduces a level of indirection imposed by the collection object. This indirection is in no way trivial and should be taken into account during implementation and, if possible, even dur-

ing design. Failure to account for this indirection can adversely impact the success of the implementation.

The elements in the collection can be either atomic or mutable objects. In the event that the elements are all of an atomic nature, there are really just two editing operations that can occur—adding or removing an element. Suppose the vendor objects in the vendors collection are meant to be atomic, i.e., they are meant to be added to or removed from the collection, but not edited directly. In such a case, the vendor collection element-accessing methods might be:

```
addVendor: aVendor

    self vendors add: aVendor
and
removeVendor: aVendor

    self vendors remove: aVendor ifAbsent: nil
```

Such collection element accessing and mutating methods are very beneficial because they allow the domain object to know when its collection is being modified.

In the event that the elements in the collection are other mutable domain objects, then each element is directly editable. Such a collection and its elements constitute part of the domain object's aggregation. With these types of collections, elements can be added to or removed from the collection as described previously, or accessed and edited directly as is shown below.

```
vendor := company vendorWithName: 'ACME'.
vendor contact: 'Franklin Black'
```

In the code above, a vendor object is accessed from a company object's collection of vendor objects. Then this vendor object is edited by changing its contact.

COPYING DOMAIN OBJECTS

It is very important that domain objects know how to copy themselves such that the copy and the original do not share any information they are not meant to share. There is nothing more frustrating than editing what you suppose to be a copy only to find out that the original has changed as well! Copying objects is a very rich topic; however, within the constraints of a single article I can only cover the highlights.

All objects know how to copy themselves because the copy method is defined in Object. The copy method, by default, only makes a *shallow copy*. A shallow copy is just a new object header and a new set of handles to the same objects referenced by the original object. For domain objects, a shallow copy falls far short of the mark. If we make a shallow copy of an employee object and edit the copy's address, we will also be editing the original employee's address! Fortunately, the copy method also initiates a *post copy* operation. A post copy includes any additional copying that might need to be conducted beyond the shallow copy. It is implemented in the postCopy method. Each type of domain object that defines instance variables should also implement its own postCopy method to define

what its copies are to look like. A postCopy method should almost always begin with the statement super postCopy. This ensures that the any instance variables defined by the superclasses are also copied appropriately. An implementation of postCopy should result in a *full copy*. A full copy can be edited without adversely impacting the original object (a full copy is not necessarily a deep copy).

In implementing a domain object's postCopy method, we must be sensitive to the type of reference made by each of the domain object's instance variables. For copying domain objects, the following guidelines apply based on the type of object referenced by each instance variable.

- Atomic objects need not be copied since they will be replaced anyway.
- Mutable objects should be copied.
- Collections of atomic objects should be copied, but their elements need not be copied.
- Collections of mutable objects should be copied, and their elements should be copied as well.

For the last guideline above, the DomainObject class provides a private method called copyCollection: that takes an original collection as an argument and returns a copy of that collection whose elements are copies of the original's elements.

COMPARING AND TESTING

Domain objects need to know how to compare themselves to other domain objects. Mostly this is for the benefit of certain collection operations such as sorting, detecting, and tests for uniqueness.

Each domain object should be able to determine if it is equal to another domain object of the same type. The exact determination of equality is strictly up to the design requirements. For instance, in one design, an Employee class might implement the method = as:

```
= anEmployee

    ^self fullName = anEmployee fullName
```

and another design might use

```
= anEmployee

    ^self ssn = aEmployee ssn
```

In the first case, the designers determined that two employees are the same if their names are the same. In the second case, the designers feel that two employees are the same if their social security numbers are a match.

Very often domain objects are presented in a sorted collection. The default behavior for a SortedCollection is to rank its elements using a less than or equals comparison. This makes it convenient to implement a <= instance method in each domain object class. Usually, a <= method conducts its comparison on the same parameters as the = method, but not always. For the above, employee objects might be ranked alphabetically as:

```
<= anEmployee

    ^self fullName <= aEmployee fullName
```

In the event that you do not want to rank a collection according to <=, you can also pre-define sort blocks to determine the ranking. Sort blocks allow the same type of domain object to have different types of ranking over different SortedCollections. A sort block takes two arguments representing two consecutive elements in the collection. The sort block's implementation describes the relationship that must hold between the two consecutive elements. The default descending ranking looks like:

```
[:e1 :e2 | e1 <= e2].
```

An ascending sort block looks like:

```
[:e1 :e3 | e1 >= e2].
```

A sort block access method for ranking employee objects by ssn would look like:

```
descedingSSN

    ^[:e1 :e2 | e1 ssn <= e2 ssn].
```

and a sort block access method for ranking by name would look like:

```
alphabeticalByName

    ^[:e1 :e2 | e1 fullName <= e2 fullName].
```

In addition to comparing itself to other domain objects, a domain object can perform several useful tests and functions on its domain information for client objects. Very common among these tests is a test for type or class. For example, the DomainObject class can implement isAddress to always return false and the Address class can implement it to always return true. This makes the isAddress message the test to ascertain if a domain object is an Address or not. There are several other problem-specific testing methods you may want to include in a domain object class testing protocol.

PRINTING AND DISPLAYING

Domain objects should not be responsible for displaying themselves on a display surface. However, they can make it easier for other objects that wish to display them. Domain objects should provide several mechanisms for representing themselves as strings, text, and even visual components. There are three categories of printing and displaying methods: print methods, display methods, and visual block methods.

The print methods consist of the printString and the printOn: methods. Every Smalltalk object knows how to respond to the message printString by returning a string that describes itself. The default implementation provided in Object is to just return the object's type, such as 'an ApplicationModel' or 'a PluggableAdaptor'. The printString message is used to describe an object in programming tools such as the Inspector and the Debugger. For the purposes of debugging and inspecting, it would be nice if the printString

message returned something a little more descriptive than just the type. For this reason, it is advantageous to customize the printString behavior for your object types. However, the printString method is not responsible for building the string. This is done in the printOn: aStream method. It is this method you want to override for your domain objects. For example, by default a Person object will return 'a Person' as its print string. We might want to override the printOn: aStream method to return something like:

```
'Person—Jones, William Robert'.
```

To do this, we would implement the printOn: aStream method for the Person class as:

```
printOn: aStream

    aStream
        nextPutAll: 'Person—'
        nextPutAll: self lastName;
        nextPutAll: ', ';
        nextPutAll: self firstName;
        space;
        nextPutAll: self middleName
```

Most domain object classes will want to implement the displayString method and perhaps other display methods. The display methods are similar to the printString method in that they return a string or text representing the receiver. The display methods are used primarily by list and table components. An object can have several different display methods—of which displayString is the default.[1]

The visual block methods return blocks called *visual blocks*. A visual block is a block that describes how an object should be represented graphically in a list, table, or notebook tab and it takes two arguments—the widget (a SequenceView, for example) and the index of the element in the collection. A visual block should evaluate to an object that understands visual component protocol.

SUMMARY

This article covered domain objects, the containers of an application's domain information. The abstract superclass of all domain objects is DomainObject. Domain objects should not reference application information nor perform application type functions. Domain objects should however: know how to copy themselves correctly, compare themselves to other domain objects, facilitate objects which print or display them, and provide comparing, testing, and other services concerning their domain information. Source code for DomainObject and examples, are available from the archives at the University of Illinois (st.cs.uiuc.edu).

Reference

1. VisualWorks List Components, June 1994.

Tim Howard is the President and Cofounder of FH Protocol, Inc. He is interested in application development using O-O technologies in general, and using the language of Smalltalk in particular. He can be reached at thoward@fhprotocol.com or by phone at 214.931.5319.

Host platform accessing framework

Multimedia: an example

Yoel Newman

THIS ARTICLE PRESENTS AN EXAMPLE implementation of the approach outlined in a previous article "An object-oriented approach to accessing external resources."[1] It covers the elements necessary for incorporating external resources such as communications protocols, database access, and multimedia services for VisualWorks 2.0. The DLL and C Connect (DLLCC) product is a prerequisite for this example.

EXAMPLE

An abstract multimedia interface with concrete support for OS/2 serves as a medium for illustrating the elements of the framework. This is only an example, and is not necessarily a full implementation of the multimedia interface in OS/2. The example is simple enough to describe in a small article and robust enough to illustrate the approach outlined in the previous article.

Specifically, the abstract interface does not include support for asynchronous communications between the API and VisualWorks. The interface only implements blocking API calls. This is an abstract interface limitation since MMPM/2 includes a mechanism for asynchronous multimedia support.

The previous article proposed a layered approach for accessing external resources (Fig. 1). At the lowest framework level lies the ExternalInterface subclass, which only makes function library calls as defined by the API. The next level consists of a two-tiered API wrapper layer. There is an abstract superclass that defines the behavior that each of the subclasses must implement. There is also a set of concrete subclasses that use the low-level ExternalInterface subclasses in their implementation to support the abstract interface. Finally there are the high-level implementation classes that encapsulate aspects of the behavior in the function library. By programming to the abstract wrapper classes' public interface, the high-level layer can use the concrete wrapper layer subclasses interchangeably.

The rest of the article will discuss each layer in greater detail.

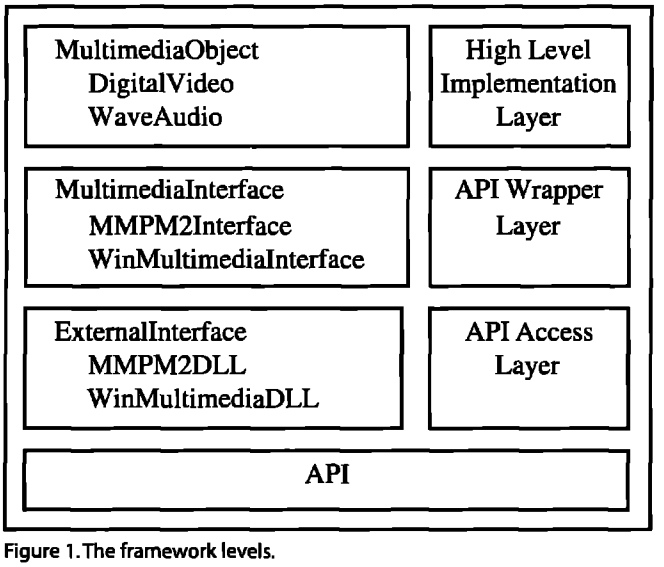


Figure 1. The framework levels.

API access layer

VisualWorks requires the ExternalInterface subclass, MMPM2DLL, to support calling the function library (Fig. 2). Parsing the file "mcios2.h" creates the definitions needed to access the multimedia features by making direct API calls. The API calls needed for this example are mciSendString and mciGetErrorString. The header file "os2def.h" is also a requirement because "mcios2.h" makes use of the standard redefined OS/2 types. For example, the redefinition of unsigned long is ULONG and the redefinition of unsigned character * is PSZ.

The following code is an example of subclassing the ExternalInterface:

```
ExternalInterface subclass: #MMPM2DLL
    includeFiles: 'os2def.h mcios2.h '
    includeDirectories: 'd:\vw20ga\mmpm2 '
    libraryFiles: 'mdm.dll '
    libraryDirectories: 'e:\mmos2\dll '
    generateMethods: ''
    beVirtual: false
    optimizationLevel: #debug
    instanceVariableNames: ''
    classVariableNames: ''
```

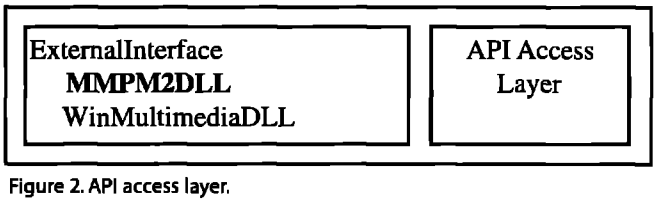


Figure 2. API access layer.

```
    poolDictionaries: 'MMPM2DLLDictionary'
    category: 'ExternalInterface-OS/2'
```

The #debug mode should be the initial optimization level. In #debug mode, function methods contain strict type-checking wrapper code. This type-checking code helps in the development and debugging of the interface class at the expense of performance. In #full mode, a significant decrease in function calls overhead occurs due to removing the type checking wrapper from the function methods.[2, p.19]

The generateMethods: keyword message takes a String as an argument. The String argument is a list of pattern-match strings. These patterns determine which external entries in the header files become compiled into Smalltalk methods.[2, pp.17,38]

For the multimedia example the string is: 'mciSendString mciGetErrorString'.

API wrapper layer

The two-tiered API wrapper layer contains an abstract interface class (Fig. 3) and the concrete subclass implementors (Fig. 4). The abstract superclass, MultimediaInterface, defines the behavior that each of the subclasses must implement. The concrete subclass, MMPM2Interface, uses the low-level ExternalInterface subclass, MMPM2DLL, in its implementation to support the abstract interface.

Both IBM and Microsoft use the Media Control Interface (MCI) as the abstract interface for their function library implementation. In this example, MCI will also serve as the abstract interface. This is an appropriate choice for the OS/2 and Windows environment. If broader platform support and code portability are requirements, then MCI may not be a suitable choice for the abstract interface. There is no guarantee that all vendors will use MCI as their abstract multimedia interface. Therefore, to cover a broader platform base, using a more generic and abstract interface whose implementation uses MCI is a better choice.

To understand the multimedia example, it is important to be familiar with the Media Control Interface. MCI provides services to applications for controlling devices in the multimedia environment. These services are available

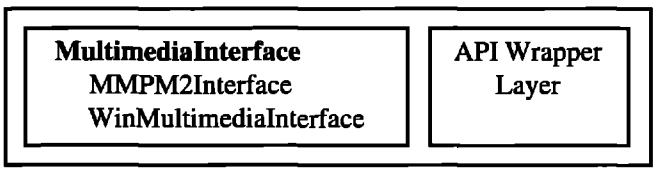


Figure 3. API wrapper layer: Abstract interface.

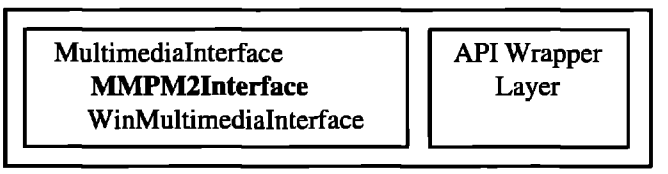


Figure 4. API wrapper layer: Concrete subclass.

through an interpretive string interface (mciSendString). The MCI string interface enables application control of media devices using textual string commands. The following are example MCI string commands:

To open the file foo.wav:	= =>	open foo.wav
To give foo.wav the alias wave:	= =>	open foo.wav alias wave
To wait for the MCI command to complete before returning from the API call:	= =>	open foo.wav alias wave **wait**
To play the alias wave from the beginning and wait for the MCI command to complete before returning from the API call:	= =>	play wave from 0 wait
To close alias wave and release any allocated resources:	= =>	close wave

The following series would open a file, play the file, and close the file:

```
open foo.wav alias wave wait
play wave from 0 wait
close wave
```

For more information about MCI, refer to either the OS/2 or Windows multimedia documentation.

The example code provides multimedia support for OS/2. Implementing support for the Windows environment requires a WinMultimediaInterface concrete subclass (Fig. 4). This subclass has the responsibility of implementing the abstract interface for the Windows environment.

The concrete subclass, MMPM2Interface, has to handle the following items in its implementation:

- Memory allocation and deallocation.
- Exception handling.
- Maintain and enforce the state of the API.

Memory allocation and de-allocation. The BlockClosure message valueNowOrOnUnwindDo: is used to handle the API call. The method evaluates the handler code whether an exception occurs or not. The main reason for using the valueNowOrOnUnwindDo: message is to free the memory allocated on the external heap. The method for freePointer is faster than the method for free. However, only non garbage-collectible pointers should receive the message freePointer:

```
mciSendString: aString
    "aString is a MCI command. The interface calls the
     connection to execute aString."
    | aRc aStringPtr |
    [ "Begin unwind block"
    aStringPtr := aString copyToHeap.
    aRc := self
        connection
            mciSendString: aStringPtr
                returnBuffer: self buffer
                returnBufferSize: self bufferSize
                callbackWindowHandle: self handle
                userWindowHandleParameter: self
handleParameter.
    "End unwind block" ] valueNowOrOnUnwindDo:
        [ aStringPtr notNil ifTrue: [aStringPtr freePointer ] ].
    aRc = self succesfulAPIRET
        ifFalse:
            [self mciGetErrorString: aRc]
```

The concrete multimedia implementation for OS/2 uses an instance variable to store a CPointer buffer. The methods for mciSendString: and mciGetErrorString: require a CPointer buffer to perform the API call. Without storing the buffer in an instance variable, the environment would incur a performance penalty because a CPointer is being allocated and then deallocated for each API call. Sometimes there is a fine line between optimization and technique. Using an instance variable to store the buffer makes sense from a design standpoint as well. Accessing the buffer using its get and set selector is preferable to passing it as a parameter in a keyword message.

```
mciGetErrorString: anErrorCode
    "Translates anErrorCode to a literal string
     representation of the error."
    | aRc anErrorString |
    aRc := self
        connection
            mciGetErrorString: anErrorCode
                returnBuffer: self buffer
                returnBufferSize: self bufferSize.
    aRc = self succesfulAPIRET
        ifTrue:
            [anErrorString := self buffer
copyCStringFromHeap.
            self class
                multimediaInterfaceErrorSignal
                    raiseWith: anErrorString]
        ifFalse:
            [self class connectionExceptionSignal raise]
```

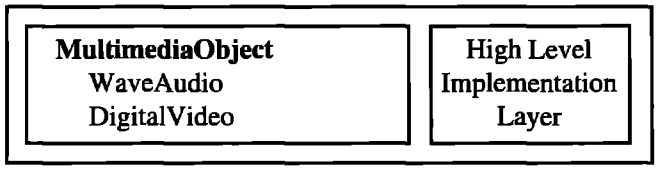


Figure 5. High-level implementation layer: Abstract interface.

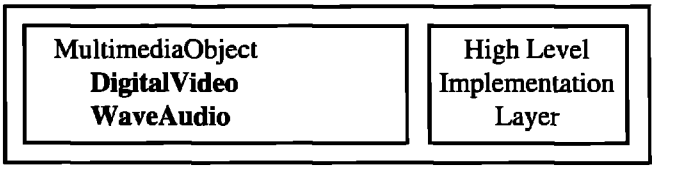


Figure 6. High-level implementation layer: Concrete subclass.

Exception handling. The concrete subclass implementors have a responsibility to handle exceptions when they occur. The implementation must also implement the hierarchy of exception handling signals defined in the abstract interface to resolve any API call failures.

The MultimediaInterface exception handling hierarchy is:

```
multimediaInterfaceSignal
    multimediaInterfaceErrorSignal
        connectionExceptionSignal
        syntaxErrorSignal
```

The method mciGetErrorString: will raise the multimediaInterfaceErrorSignal for an unsuccessful API call. The textual representation for the error code becomes an exception parameter when raising the exception. If the function mciGetErrorString fails, the method will raise the exception connectionExceptionSignal.

Maintain and enforce the state of the API. There are many approaches to handling state within the API wrapper layer implementation. Two possible approaches are:

1. Maintain the state of the API using an instance variable. In this approach, the concrete implementor has the responsibility for maintaining and enforcing the state. The drawback to this approach is that it relies heavily on case-style statements in its implementation.
2. Use the State pattern by implementing concrete support using multiple subclasses representing the different states of the API. The State pattern will "allow an object to alter its behavior when its internal state changes. The object will appear to change its class."[3]

For this example, concrete support for MMPM/2 would include the classes:

```
MultimediaInterface
    MMPM2Interface
        MMPM2State
            MMPM2Reset
            MMPM2Initialized
            MMPM2Loaded
```

The MultimediaInterface class declares an abstract multimedia interface. The MMPM2Interface class maintains a state object (an instance of a subclass of MMPM2State) and delegates all state-specific requests to this state object. Subclasses of MMPM2State implement state-specific behavior particular to the specific state of the interface.[3]

For simplicity, this example uses approach 1—using an instance variable to maintain and enforce state.

High-level implementation layer

The two-tiered high-level implementation layer defines a class for handling the high-level abstract multimedia behavior (Fig. 5) and concrete subclass implementors for handling the specific multimedia behavior. The abstract class, MultimediaObject, defines the abstract behavior for all multimedia objects. The concrete subclasses

DigitalVideo and WaveAudio (Fig. 6) provide the specific behavior for digital video and wave audio.

The MultimediaObject subclasses hold an instance of a MultimediaInterface subclass in the instance variable, interface:

```
play: anMCICommand
    "play the device, close the file on error."
    MultimediaInterface multimediaInterfaceErrorSignal
        handle:
            [:exception |
            self interface state == #open
                ifTrue: [self interface close] ]
        do:
            [self interface
                command: anMCICommand;
                open;
                play;
                close]
```

One approach to establishing the correct platform interface is to do so at runtime. The MultimediaObject subclasses detect which platform is running and set their interface to the appropriate instance of a MultimediaInterface subclass. Another approach to establishing the class of the interface is to use a class initialization method that either stores or sets the current platform. These types of techniques provide basic elements of platform independence.

```
defaultMultimediaInterface
    self platform isNil
        ifTrue: [self initialize].
    self platform = = #os2
        ifTrue: [^MultimediaInterface os2]
```

This example uses lazy initialization for the platform if not currently set:

```
initialize
    "Initialize the platform"
    Platform := OSHandle currentOS
```

The current platform is available using:

```
ExternalInterface currentPlatform   #(#os2 'os2 OS/2
                                      V2.30')
OSHandle currentOS                  #os2
OSHandle currentPlatformID          'os2 OS/2 V2.30'
```

In the current implementation, the classes DigitalVideo and WaveAudio do not contain any significant implementation differences. The reason the implementations are in separate subclasses is that DigitalVideo supports the display of video in a user defined window. Support for digital video display in a user defined window requires minor modifications to the DigitalVideo implementation. The WaveAudio implementation requires minor modification to support use of non-blocking, non-notification, API calls.

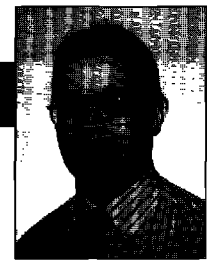
SUMMARY

Smalltalk is a pure object-oriented environment. At first, it may seem the use of function libraries is incompatible with accepted Smalltalk idioms. However, using an object oriented approach to accessing external resources creates a higher level implementation that extends the development environment interface. This article and the preceding article (Fig. 1) have discussed and demonstrated an object-oriented approach for incorporating function libraries into Smalltalk.

References

1. Newman, Y. and M. Parvin. An object-oriented approach to accessing external resources, SMALLTALK REPORT 4(7), 1995.
2. ParcPlace Systems, VISUALWORKS DLL AND C CONNECT USER'S GUIDE, 1994.
3. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1994.

Yoel Newman is a Senior Systems Consultant and can be reached at yoel_newman@aol.com.



Math

Alan Knight

LAST ISSUE WE DISCUSSED some of the problems with using floating point numbers, particularly in calculations involving money. Far too many people seem to be unaware of the difficulties involved and very surprised to find round-off errors in their calculations. In this installment, we're going to look at some of the alternatives.

FIXED POINT

A more suitable representation for money is fixed-point numbers. These support a fixed number of places after the decimal point, and an unlimited number in front of it. Several Smalltalk implementations already support fixed-point, and in those that don't it's not difficult to implement them.

There are two obvious implementations for fixed-point numbers. The first is as integer numbers of some smaller unit. For example, to represent money, you might determine the smallest unit you need to be accurate to (e.g., 1/100th of a cent) and do all your computations in this unit.

This representation works for addition and subtraction, and is quite reasonably fast, but it has problems multiplying and dividing. It also has problems if you have fixed-point numbers with different degrees of precision.

One way to get around these limitations is to use fractions, as ParcPlace does for their fixed-point implementation. Their explanation, taken from the class comment, is as follows:

> **There are two possible ways to express FixedPoint numbers. One is as a scaled Integer, but the problem here is that you can lose precision during intermediate calculations. For example, a property that seems useful is that the calculation (1.000 / 7 * 7) should give you back the number 1.000.**

Fractions are accurate, because they can represent any rational number, but are very slow compared to scaled integers.

Coercion

Fixed-point in either of these implementations gives us accuracy, but it's a bit of work to ensure that all calculations are done in fixed point. As X. Alvarez points out:

> **FixedPoints are coerced...against Floats or Doubles due to the higher generality...So my FixedPoint numbers aren't protected against being coerced to higher generalities...invalidating the "100.9f - 100.0f" approach because I can't be sure that no float or double sneaks in.**

This problem can also arise if you just want to use double precision. Curt Welch (curt@to.mobil.com) writes:

> **...we had to hunt down every floating constant in the code and make it a double....And there's no easy way to be sure we found them all.**
>
> **And don't make the mistake of thinking that just because the variables are set up correctly as doubles that you are safe...For example:**
>
> **a := 10.0d.**
>
> **a * 0.1 produces: 1.0000000149012d (a loss of 6 places of accuracy).**

The loss of accuracy occurs because the single-precision constant is converted to a double, but it still has the accuracy of a single-precision number. The result has much less accuracy than expected. This is a nasty problem, because it's hard to ensure that nobody has forgotten the "d" (double) or "f" (fixed-point) on the end of a numeric constant.

Changing the default

What we'd really like to do is to change the way the system treats numbers. For ParcPlace, Douglas Johnson (doug@rwi.com) writes:

> **In ParcPlace VW2, change the method readSmalltalkFloat:from: on the class side of Number...There is a line about 18 lines down that reads "coercionClass := Float." change it to "coercionClass := Double."**

The same technique can be used to make the default numeric class Fixed. Similar techniques should work in other dialects, although it's possible that in some versions the relevant source code isn't available.

I don't think I'd recommend this technique in general, as it can still break down in some rather tricky ways. When you make this change you are effectively changing the compiler. This will change the default class of numbers from now on, but it doesn't affect code that was compiled before the change was made. Any existing methods (including system methods) that have float constants in them will still use regular floats. You'll probably need to recompile any system methods with floats in them. Here's a piece of code for ParcPlace that does this automatically.

```
CompiledMethod allInstances do: [:eachMethod |
    | classAndSelector floatLiteral |
    classAndSelector := eachMethod who.
```

Alan Knight is a significant figure at The Object People, 509-885 Meadowlands Dr., Ottawa, Canada, K2C 3N2. He can be reached at 613.225.8812 or by email as knight@acm.org.

```
floatLiteral := eachMethod allLiterals
    detect: [:eachLiteral |
        eachLiteral class == Float]
    ifNone: [nil].
(classAndSelector notNil and: [floatLiteral notNil])
    ifTrue: [
        classAndSelector first compilerClass
            compileClass: classAndSelector first
            selector: (classAndSelector at: 2)]].
```

Even that isn't necessarily enough. ENVY stores compiled code in its database, and any references to floats in that compiled code need to be changed (and I think the current version of ENVY introduces a bug in allLiterals). There might also be global or class variables holding numeric constants or even blocks compiled before the compiler change.

Furthermore, we're only changing the default class. Code that specifies a particular numeric type will still compile to produce that type. For example, Float>>pi might return 3.14159265358979d, which will be a double no matter what the default numeric class.

Application-specific numbers

As we've seen, dealing with numbers can be very tricky. Fortunately, this is Smalltalk, and if none of the built-in numeric types are satisfactory we can build our own and integrate them into the system. These could be additional numbers (e.g., complex) but they are more likely to be application-specific quantities that have additional behavior beyond simple arithmetic. In a sophisticated application, we are likely to need a Money class that knows the appropriate behavior for money objects. This may need to handle a number of complex issues that don't apply to standard numbers. These include conversion between different currencies and existing rules for how round-off should be handled. Tom Stambaugh (tms@stambaugh.com) writes:

> **If you or your client is serious about currency and money manipulation, you need to familiarize yourself with existing literature and work in the field. This is not as easy or as obvious as you might think, and has (not surprisingly) been the subject of much work and standardization efforts. If your client is an SEC regulated institution, you will also probably have to show that your work complies with relevant regulations.**
>
> **Probably, the most straightforward way to accomplish all this is to get your hands on an external math package with the needed support, and wire wrappers into the Magnitude family. While its certainly possible to "roll your own," and a number of the suggestions here are quite reasonable, its also possible to code yourself and your client into some *very* deep ratholes.**

Tony Law (tlaw@cix.compulink.co.uk) adds:

> **...there are some currencies where the base unit is small value where local legislation requires integer calculation (Belgian Franc and Lira for example)....BT here calculates phone bills using fractions of 1p in unit costs, calculates VAT on the lot, then rounds the answer. This would not be allowed in Italy, I believe (anyone from Belgium or Italy care to comment)?**

A Money class (or classes) is a good way of dealing with these issues. It should not, of course, inherit directly from a concrete numeric type (like Integer or Float) but from an abstract class higher in the Magnitude hierarchy. Unfortunately, money objects still don't make everything work automatically under all circumstances. If I mix money and other numeric types, I may still lose precision the same way as I would mixing floats and doubles. For example:

```
(Money new amount: 1.0) + 0.1
```

will have to deal with the inherent inaccuracy in 0.1. If that inaccuracy is less than the precision of the money object (as it would be in this case) the float could simply be rounded to the appropriate precision. If not, it either has to raise an exception or introduce some inaccuracy into the calculation. The real advantage of money objects is that I have control over these decisions and can act in a manner appropriate to the application.

There are lots of other extended numeric types that might be useful in particular applications

- Numbers that automatically keep track of accumulated inaccuracy, providing ranges instead of exact results (e.g. 1.53 ± 0.24)

continued on page 19

Queries in Smalltalk

Jay Almarode

At some point in just about any large application, there is a need to search over a collection to find all the objects that match some criteria. With Smalltalk, users have a computationally complete, extensible language in which to express these queries. Every collection class provides a means to iterate over its contents, allowing any kind of complex behavior to be executed on each element. For example, inside the argument block of the do: method, an application developer can send any desired message to each element in the collection, performing navigation through the network of objects until some discriminating message is sent to determine whether an object should be included in the result or not. Class Collection provides a default implementation of three convenience methods to help in composing these query results. The methods select:, reject:, and detect: are understood by all collections, and subclasses may reimplement them to provide a more optimized implementation.

Smalltalk users have a complete, extensible language in which to express queries.

The semantics of select:, reject:, and detect: are easily understood by looking at their default implementation in class Collection. For example, the select: method iterates through the collection (using the do: method, which is implemented by subclasses), and evaluates the argument block for each element. If the block evaluates to true, the element is placed in the result. One thing to notice is the kind of object that is returned as the result of the query. The select: and reject: methods are defined to return an object similar to the receiver. I say "similar" because the responsibility of determining the appropriate kind of return value actually belongs to the receiver. In SmalltalkDB, the speciesForSelect method returns the kind of object that should be constructed for select: and reject: queries. In VisualWorks and Visual Smalltalk, the method that does this is called "species". The default implementation of this method returns the same class as the receiver. Therefore, if you send a bag the select: message, you will get a bag as the result. Subclasses may override the speciesForSelect method when it is desirable to answer an object of a different class than the receiver. A later example will show how this distinction can be useful for certain kinds of queries.

The detect: method is often used when one wants to ask about the existence of a condition among the objects in the collection. For example, if you wanted to ask "Are there any employees whose 401k contributions are greater than 10% of their salary?" you might execute the following:

```
aSetOfEmployees detect: [ :emp |
    emp 401kContribution > (emp salary * 0.10) ].
```

This would return the first employee object encountered that matched the condition. The detect: method could also be used for a lookup operation when a user believes there is only one object in the collection that satisfies the search criteria. However, this usage is ill-advised because it would seem to work correctly even if there was more than one object that matched, possibly hiding data inconsistency problems. In this scenario, it is probably best to use a hash dictionary or some other collection that provides optimized access by key. Another alternative is to use the select: message and explicitly test the size of the result.

In SmalltalkDB, the mechanism for querying collections has been extended with a different kind of block, called a SelectBlock. A SelectBlock, which is delimited with { and }, restricts the kind of statements inside the block to be conjoined predicates of a certain form. This form allows the use of a dot notation to specify a path of named instance variables to traverse. I will not get into a lengthy syntactical description of this path notation; most Smalltalk programmers should find it straightforward after viewing a few examples. Using this notation, for example, one could ask the query "Give me all employees whose address is at a given zip code and who are older than their spouse" by executing the following:

```
aSetOfEmployees select: { :emp |
    (emp.address.zipcode = 97223) & (emp.age >
emp.spouse.age) }
```

The use of SelectBlocks has a number of advantages. The main advantage is that the execution of the SelectBlock is

Jay Almarode can be reached at almarode@slc.com.

handled by a subsystem that can utilize indexes and standard query transformation techniques to execute queries faster. For large collections, iterating through the entire contents may be prohibitively slow. In SmalltalkDB, subclasses of AbstractBag can have indexes created along any number of paths to speed up queries. An index is used to avoid brute force iteration by utilizing auxiliary structures (B-trees and hash dictionaries) to perform fast searches. Another advantage of using SelectBlocks is the conciseness of specifying queries using the dot notation. When specifying a path traversal using the dot notation, the programmer does not have to worry about paths that cannot be traversed (ie, a nil value is reached before the end of the path). Objects that cannot be traversed the entire length of the path are not considered when constructing the query result. This can save writing some tedious code that checks whether the value of an instance variable is nil before proceeding along the path. If you write production code that does not perform this kind of checking, you risk getting a "Message not understood" error when the application is running.

With the latest release of GemStone, the dot notation for SelectBlocks has been extended to allow paths that traverse through instance variables whose value is a kind of bag (including sets). This means that the value of an instance variable along the path may be any kind of bag, and the remainder of the path is traversed for all elements contained in the bag. An instance variable whose value is a bag is indicated by using an asterisk as one of the terms in the path expression. For example, to pose the query "Give me all employees with children who are 18 years or older" one could execute the following:

```
aSetOfEmployees select: [ :emp | emp.children.*.age >= 18 ].
```

In this example, aSetOfEmployees contains instances of class Employee with a named instance variable children whose value is a set. Each set of children may contain instances of Person (the superclass of Employee), which has a named instance variable age whose value is an Integer. When evaluating the query, the asterisk in the path expression indicates that the next object along the path is a bag, and the path traversal is continued for all elements in the collection.

As mentioned earlier, the result of a select: or reject: query is an object similar to the receiver, depending upon the receiver's implementation of speciesForSelect. This distinction can be useful if the query result is a bag, which allows more than one occurrence of the same instance, rather than a set, which only allows one occurrence. If the query result is a kind of bag, then a particular object's number of occurrences in the result is equal to the number of times the object satisfies the query. In the previous example, if the result of the query contains three occurrences of the same employee, then that particular employee has

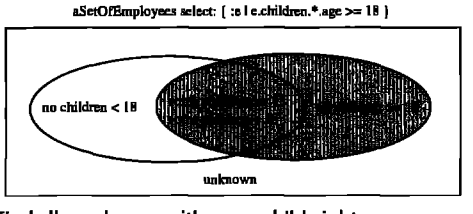


Figure 1. Find all employees with some child eighteen years or older.

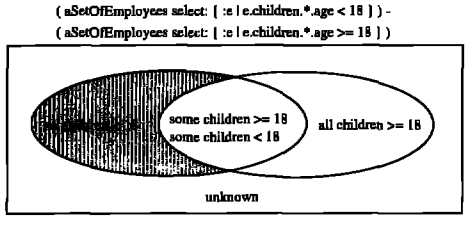


Figure 2. Find all employees with no child eighteen years or older.

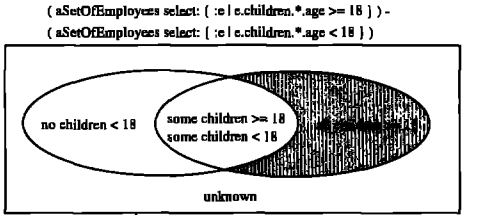


Figure 3. Find all employees with all children eighteen years or older.

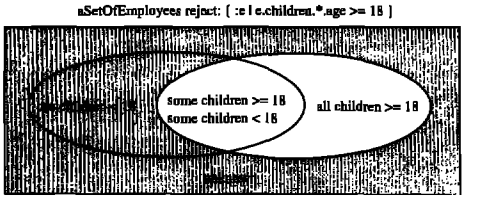


Figure 4. Find all employees with no child eighteen years or older (using reject).

three children who are 18 years or older. In building applications, if the receiver is a kind of set, but you desire the query result to be a kind of bag, then you can create a subclass of Set, override the speciesForSelect method to return a bag, and use the new subclass to hold your employees.

In SmalltalkDB, a query through a path that includes a bag is defined such that an object satisfies the query criteria if any subobject in the traversed bag satisfies the query. In the previous query, an employee is contained in the query result if he or she has *any* child 18 years or older. Given these semantics, it is fairly easy to pose other kinds of common queries. One way to characterize these common queries is according to whether some, none, or all of the subobjects in the traversed bag must satisfy the selection criteria. The Venn diagrams in Figures 1–3 illustrate the partitioning of objects for each kind of query. The diagrams divide the universe of employees into four groups: (1) those employees with *no* children who are 18 years or older, (2) those employees with *some* children younger than 18and some children 18years or older, (3) those employees with *all* children 18 years or older, and (4) those employees for which we do not know because the path cannot be traversed to the end.

Now let's see how we can express each kind of query using the semantics described above for querying through a path containing a nested bag. Figure 1 illustrates the query asking for all employees with *some* child 18years or older, indicated by the shaded region. In this case, the semantics of querying through a path with an asterisk gives the desired answer. Figure 2 illustrates the query asking for all employees with *no* children that are 18 years or older. This is achieved by taking all employees with any child younger than 18 and using the difference operator to subtract out all employees with any child 18 years or older. Finally, Figure 3 illustrates the query asking for all employees with *all* children 18 years or older. This is similar to the second query except that we start with all employees with any child 18 years or older and subtract out those employees with any child younger than 18 years.

In the previous examples, it might seem that reject: could be used for the second and third queries. However, for SelectBlocks, reject: is defined as answering the difference between the receiver and the result of select: with the same query block. In other words, aBag reject: aSelectBlock is equivalent to aBag - (aBag select: aSelectBlock). This means that using reject: will include objects that could not be traversed entirely along the query path, because those objects are excluded when select: is used. Figure 4 illustrates the difference when reject: is used for the second query above. If your application ensures that all objects can be traversed along the entire path, then using reject: is equivalent to the second query above.

Hopefully this column has enlightened you to the flexibility and power of using Smalltalk as a query language. My next column will discuss the use of indexes to speed up queries and how indexes can be extended for user-defined classes.

COMP.LANG.SMALLTALK *continued from page 16*

- Numbers with associated units of measurement ($27.50, 300,000 km/s, 6.2 MWh)
- Numbers whose degree of precision can adapt to a particular calculation. I might define a number as the square root of two. If involved in a calculation, it could attempt to determine the accuracy of the other number(s) involved and compute a finite-precision representation of itself to as many digits as required.

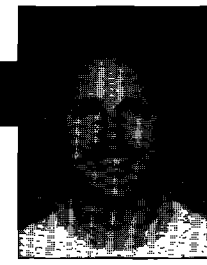
Smalltalk's ability to define new numeric classes that can interact transparently with the basic numbers opens up an enormous range of possibilities. I've only scratched the surface of this very interesting area.

Floats for money

Ultimately, what you need from numeric classes is determined by the particular needs of your application. General principles are often wrong when applied to exceptional circumstances. A good example comes from Curt Welch (curt@to.mobil.com), who writes:

> **I'm working on a financial system in Smalltalk and all our money values are floats and doubles. I wouldn't think of using some type of integer. The difference is that our system is not an accounting system. It's a risk analysis system. We aren't calculating account balances, we are estimating the value of a portfolio (and how that value may change over time.)**

The only really consistent rule is that you need to be careful. Assuming that numeric types in a computer work the same way as basic mathematics is dangerous in any language.



Kent Beck

Clean code: Pipe dream or state of mind?

Is it my imagination, or are these columns getting harder to write? I think I know exactly what I want to say, but I've started writing three different times without getting anywhere. Maybe this third time will work.

Simply put, here's what I want to say—the best programming style for Smalltalk is to have lots of little methods, and lots of little objects.

That's a pretty broad statement, broad enough that it can't possibly be true in all cases. What are the trade-offs, the issues that affect programming style?

Why do I care? Why not just let a thousand different styles blossom? Here's what I've done over and over. I'll be asked by a client to help them figure out what's going wrong with a piece of code. The first thing I'll do is reformat the code in question so I can follow the flow of control. Then I'll start breaking big methods into smaller pieces, asking the client to name the new methods I create.

At some point in this process the problem becomes obvious. The proposed name doesn't match what the method is doing. A computation that should happen once is happening twice. A computation that should be happening on only one side of a conditional happens on both.

I never get over feeling that a problem like this, where the solution is merely to clean up, didn't need to happen in the first place. It's not like what I do is profound—I don't have to go away and think hard. I mechanically apply a few simple patterns. The answer appears. I'm not there to give deep advice. I just provide permission.

Here are the important patterns for this kind of debugging:

- **Composed Method.** Give each method one simple job. Keep all the operations in the method at the same level of abstraction. This naturally results in many methods, most of them a few lines long.
- **Explaining Temporary Variable.** Communicate the sense of a complex expression by pulling a subexpression out and assigning its value to a variable named for the meaning of the subexpression.
- **Indented Control Flow.** For messages with two or more parameters, put each keyword and its argument on its own line, indented one tab. This makes multi-keyword messages easy to spot and to read.
- **Rectangular Block.** Start blocks with two or more lines on a fresh line, indented one tab. This makes the shape of the control structures easy to scan.

Why don't these clients keep their code clean themselves? Why do I have to step in for them to do what is obviously (to me) the right thing to do?

Here are some reasons I've heard:

- **"I don't have time."** Folks will spend half a day working on a bug, trying various fixes without success. Often, 15 minutes of cleanup makes the problem obvious and improves the code for the future at the same time. Even if you don't find the bug right away, you'll be in a much better position to fix it when you do find it if the code is clean.
- **"I don't know how."** It might take a while to get accustomed to the patterns above, but a few hours investment will pay off for years. If you don't agree with the details of the patterns, if you indent code differently, that's fine, but do it *some* way. Life is too short to continually make detailed coding decisions.
- **"It's not important."** The cost of a piece of code over its many-year life is dominated by how well it communicates to others. If it is easy to understand, it will cost your company less while bringing the same benefits.
- **"It's the wrong thing to do."** Some people claim that many small methods and many small objects are harder to understand than fewer bigger objects and methods. Software engineering is all about mapping intention to implementation, moving from *what* to *how*. Every method name, every class name is an opportunity for you to communicate what is happening. Every method body and the code in every class is the means by which you specify how it is to happen. Big methods and big objects mean you are focused on how, not what.

I finally realize why this has been so hard for me to write. I'm frustrated. I keep explaining the principles of quality code over and over, and I keep getting the same arguments. I'm sure this reflects more on me than on anyone else, but I'm still frustrated.

TRUE CONFESSIONS

Confession being good for the soul, and all moralizing aside, do I really always keep my own code squeaky

Kent Beck has been discovering Smalltalk idioms for ten years at Tektronix, Apple Computer, and MasPar Computer. He is the founder of First Class Software, which develops and distributes developer tools for Smalltalk. He can be reached at First Class Software, P.O. Box 226, Boulder Creek, CA 95006-0226, 408.338.4649 (voice), 408.338.3666 (fax), or by email at 70761,1216 (Compuserve).

clean? I like to think so, and for the most part it's true, but every so often reality comes up and smacks me in the face (thanks, Reality, I needed that). Here's a nasty incident from my recent past that illustrates the value of clean code and how I sometimes resist it.

Original code

I've been working on a new-from-scratch version of HotDraw, the graphic editor framework I wrote with Ward Cunningham at Tektronix lo these long and many. Anyway, here is the code that gets invoked when the mouse button goes down and the editor is in selection mode.

```
SelectionTool>> button1Down: aPoint
    self originalPoint: aPoint.
    self previousPoint: aPoint.
    self figure: (self drawingPane figureAt: aPoint).
    self figure isNil
        ifTrue: [self drawMarquee]
        ifFalse:
            [(self selectedFigures includes: self figure)
                ifTrue: [^self].
            self resetSelections.
            self selectFigure]
```

There are two cases—if the mouse is over a Figure when the button goes down, the Figure should be selected. Otherwise, this should start group selection. This method is only eight lines long, it reads okay, so what's the problem? Well, it certainly violates the rule that a method should do one job. I wasn't satisfied, but it worked okay so I left it alone.

Here's the code for when the mouse moves while the button is down:

```
SelectionTool>> button1Move: aPoint
    | delta |
    self previousPoint: aPoint.
    self figure isNil
        ifTrue: [self moveMarquee]
        ifFalse:
            [delta := aPoint - self previousPoint.
            self selectedFigures do: [:each |
                each moveBy: delta]]
```

If we are selecting a group, track the mouse. If we are moving a Figure (or actually all the selected Figures), move them. Now I begin to get glimmerings of what is wrong. The conditional code "self figure isNil ..." is repeated. Let's look at the "button up" code.

```
SelectionTool>>button1Up: aPoint
    self figure notNil ifTrue: [^self].
    self drawMarquee.
    self selectAll: self selectedFigures
```

Here the same conditional appears, but in a different guise. I worked with this code for about a month never realizing how hard it was to manipulate until I added Handles. Handles are like Figures because they live in the Drawing, but they are like Tools because they interpret input. When the selected Figure is a Handle, the Tool doesn't do anything itself, it just passes the input along to the Handle.

I started to extend the code above to implement the case where the mouse is over a Handle. It wasn't going well so I finally took a step back and asked myself "why?"

One simple change I could make is adding an "isSelectingGroup" method:

```
SelectionTool>>isSelectingGroup
    ^self figure isNil
```

I could replace the tests in the three input methods above so they read better. Then I could add a "shouldDelegateInput" method so I could tell if the Tool should delegate input:

```
SelectionTool>>shouldDelegateInput
    ^self figure acceptsInput
```

However, this doesn't solve the deeper problem, which is the repeated conditional code. All good programming style codes down to this: say everything once and only once. Having the same conditional code in three methods violates this rule.

State Object

State Object is the pattern for eliminating repeated conditional code and adding flexibility at the cost of additional objects and messages. Here's how I did it. First I created SingleSelectionState and GroupSelectionState:

```
Class: SingleSelectionState
    superclass: Object
    instance variables: figure previousPoint

Class: GroupSelectionState
    superclass: Object
    instance variables: originalPoint previousPoint
```

Then I gave them each their portion of each of the three input methods. The instance variables figure and previousPoint moved from the Tool to the SingleSelectionState. The variables originalPoint and previousPoint moved from the Tool to the GroupSelectionState. The messages Tool>>select Figure and Tool>>drawMarquee have to take an additional parameter because the Tool no longer stores these variables directly.

The way I added these methods was to mechanically copy each of the SelectionTool input methods to each state, delete the parts that didn't apply to that state, and then change messages to "self" into messages to "aTool" where necessary:

```
SingleSelectionState>>button1Down: aPoint for: aTool
    self previousPoint: aPoint.
    (aTool selectedFigures includes: self figure)
        ifTrue: [^self].
    aTool resetSelections.
    aTool selectFigure: self figure
```

```
GroupSelectionState>>button1Down: aPoint for: aTool
    self originalPoint: aPoint.
    self previousPoint: aPoint.
    aTool drawMarquee: self marqueeRectangle

SingleSelectionState>>button1Move: aPoint for: aTool
    | delta |
    delta := aPoint - self previousPoint.
    self previousPoint: aPoint.
    aTool selectedFigures do: [:each | each moveBy: delta]]

GroupSelectionState>>button1Move: aPoint for: aTool
    aTool drawMarquee: self marqueeRectangle.
    self previousPoint: aPoint.
    aTool drawMarquee: self marqueeRectangle

SingleSelectionState>>button1Up: aPoint for: aTool
    "Do nothing"

GroupSelectionState>>button1Up: aPoint for: aTool
    aTool drawMarquee: self marqueeRectangle.
    aTool selectFiguresIntersecting: self marqueeRectangle
```

Invoking the state

Now I had to set up the right state in the first place:

```
SelectionTool>>setSelectionState: aPoint
    | figure |
    figure := self drawingPane figureAt: aPoint.
    self state: (figure isNil
        ifTrue: [GroupSelectionState new]
        ifFalse: [SingleSelectionState figure: figure])

SelectionTool>>button1Down: aPoint
    self setSelectionState: aPoint.
    self state
        button1Down: aPoint
        for: self
```

The other two SelectionTool input methods delegate to the current state:

```
SelectionTool>>button1Move: aPoint
    self state
        button1Move: aPoint
        for: self

SelectionTool>>button1Up: aPoint
    self state
        button1Up: aPoint
        for: self.
    self clearState
```

Handles

Now adding support for Handles is easy. First I add a new state that delegates to its Figure:

```
DelegationSelectionState
    superclass: Object
    instance variables: figure previousPoint
```

I make sure I create one of these states when the mouse goes down over a Handle:

```
SelectionTool>>setSelectionState: aPoint
    SelectionTool>>setSelectionState: aPoint
    | figure |
    figure := self drawingPane figureAt: aPoint.
    self state: (figure isNil
        ifTrue: [GroupSelectionState new]
        ifFalse:
            [figure acceptsInput
                ifTrue:
                    [DelegationSelectionState figure: figure]
                ifFalse: [SingleSelectionState figure: figure])
```

The input methods in the DelegationSelectionState delegate to the Figure:

```
DelegationSelectionState>>
    button1Down: aPoint for: aTool
        self previousPoint: aPoint.
        self figure
            button1Down: aPoint
            for: aTool

DelegationSelectionState>>
    button1Move: aPoint for: aTool
        self figure
            button1MoveBy: self previousPoint - aPoint
            for: aTool.
        self previousPoint: aPoint


DelegationSelectionState>>button1Up: aPoint for: aTool
    self figure
        button1Up: aPoint
        for: aTool
```

CONCLUSION

What can I conclude from all this?

1. Simple code is its own reward. When you're stuck, try cleaning up first. Chances are you'll get out of your jam more quickly, and your code will be a better place to live later.
2. Use simple rules. Cleaning up code is simple. Don't try to change the behavior while you are cleaning up. If you spot a mistake, wait until a reasonable stopping spot before fixing it.
3. These new robes are a bit breezy. Don't worry if everything isn't clean all the time. It isn't for me, nor do I think it should be. Progress implies chaos, at least for a while. Make sure you clean up afterwards, though.

Controlling coupling

Mark Lorenz

THERE IS ACCEPTABLE AND NECESSARY COUPLING between objects, and then there is unacceptable and unnecessary coupling. The latter coupling results in more brittle systems and correspondingly higher development and maintenance costs.

So, which is which? The Law of Demeter defines acceptable coupling as messaging to:

- self or super
- your class
- an object you create
- an object passed to you as a parameter.

Similarly, the law defines unacceptable coupling as messaging to:

- globals
- objects returned from other messages.

Globals are fairly obvious, because they are available to every object in the system—a change to a global can ripple through the entire system! But why not send messages to returned objects? Lets take a look at an example Law of Demeter violation:

```
Account
withdraw: anAmount
        "subtract anAmount from my balance"

    | newBalance |
    ( anAmount < self balance )    "OK—a parameter
                                    passed to me"
        ifFalse: [ "error handling for negative values" ].
    super loanPayment: anAmount. "OK—my superclass"
    newBalance := self balance—anAmount.
    ( newBalance > 0.0 ) "OK—I created newBalance"
        ifTrue: [ self owner transactions add: anAmount ]
            "NOT OK! I have assumed an implementation
                for my owner's transactions collection"
        ifFalse: [ self checkOverdraft.].  "OK—my service"
```

This code lists a method that has an example of bad coupling: the add: method. A better way to design this would be for the Account class' owner object to be responsible for maintaining Transactions, with an addTransactionFor: method that would accept anAmount as a parameter. This would keep design decisions more localized and therefore easier to maintain.

Lets take a look at an example model (Fig. 1) to see what coupling results when we use sequences of message sends.

Figure 1 shows a piece of an object model for a retail Store, with some containership relationships shown. Let's say we have the following client code for this model:

```
( self store customers ) do: [ :eachCustomer |
    ( eachCustomer salesTransactions )
        do: [ :eachTransaction |
        ( eachTransaction lineItems ) do: [ :eachLineItem |
            ( eachLineItem product = aProduct ) ifTrue: [
                aCollection add: eachCustomer. ].
        ].
    ].
...
```

This code marches across the object relationships, leveraging detailed design choices to access objects across the business model. So what happens when the design changes? The client code is at greater risk of breaking with the coupling that has been designed into the system.

The following code example shows a better design that has less coupling:

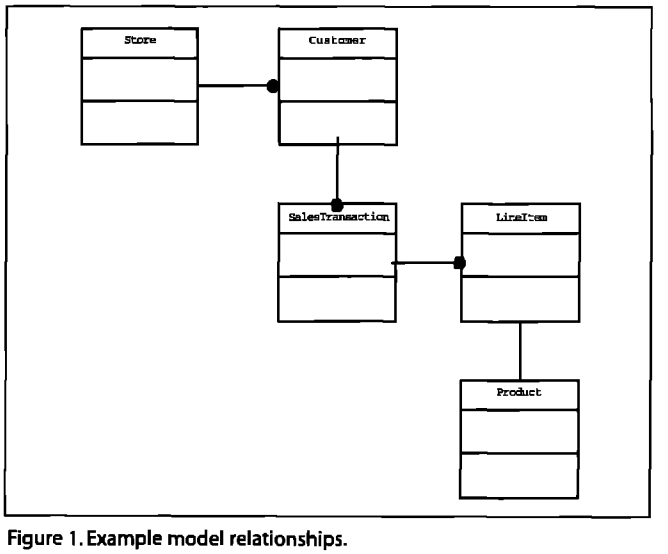


Figure 1. Example model relationships.

Mark Lorenz is Founder and President of Hatteras Software Inc., a company specializing in O-O project management, design quality metrics, rapid modeling, mentoring, and joint development to help other companies use object technology effectively. He welcomes questions and comments via email at mark@hatteras.com or voice mail at 919.319.3816.

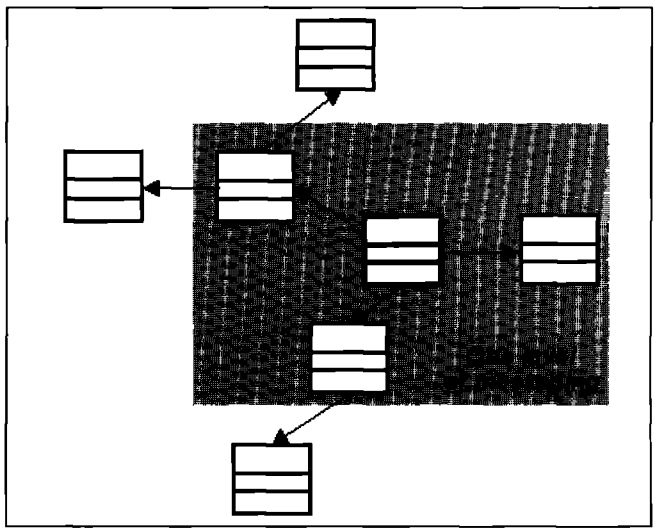


Figure 2. Layers of knowledge surrounding an object.

```
( self store purchasersOf: aProduct ) do: [ :each |
    self mailSaleFlyerTo: each.
    ].
```

This design requires that the Store support a purchasersOf: method. Other classes may be required to support additional public methods as well. The benefits for this additional cost in development time are:

- this code is far less likely to break if an implementation is changed
- greater levels of reuse and robustness are possible for *all* clients of the business model objects

A design goal should be to keep your objects as self-managing as possible, reducing coupling to other objects. Figure 2 indicates a boundary that is one layer out from an object, delimiting the relationships that can most safely be leveraged to get work done. Certainly, your systems will not be this cleanly defined in all cases, but this is a goal to keep in mind while you are developing your O-O system.

SUMMARY

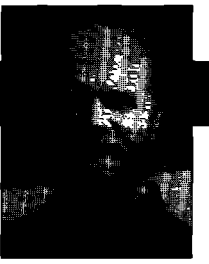
We have examined good and bad types of coupling in O-O systems, and the resulting effects of the different ways we design our systems. In general, we can achieve higher levels of reuse and robustness while simultaneously reducing maintenance costs by restricting the coupling in our systems.

Terminology

- **coupling**: Knowledge of another object's model relationships and/or design choices, usually indicated by messaging to that object.
- **object model**: Objects and their relationships required to represent your business domain and business rules.

References

1. Sakkinen, M. The law of demeter and C++, SIGS PLAN NOTICES 23(12):38, 1988
2. Lorenz, M. RAPID SOFTWARE DEVELOPMENT, SIGS Books, New York, 1995.



Jan Steinman Barbara Yates

Managing project documents

SMALLTALK IS GROWING UP. It's rapidly moving out of the research lab environment into production environments involving large teams, with the requisite procedures, standards, conventions, and bureaucracy. Through years of introducing Smalltalk into organizations, we've noticed that cultural and procedural issues have more impact on success than technical issues.

In this column, we'll be bringing you tools and techniques we've found useful in the broadest sense of "managing objects." If you are involved with Smalltalk (or hope to be involved), and are a manager or technical leader (or hope to be one!), we hope to be addressing many of your project-related concerns.

THE DOCUMENTATION PROBLEM

Success at last! It's been tough—you managed to put a good team together, train them, fight the "why aren't you using C++?" kinds of battles, and still get your project done in record time. It has amazingly few problems for a new project, and the alpha users love it, so you take it to your department head, who schedules a meeting for final release approval.

Waiting at the release meeting is an old political enemy, who at first opportunity says, "So, the project is documented in accordance with MegaCorp Standards and Procedures Manual section 32, subsection C?"

"Well, not really, you see, because it's, well, new technology, and things have to be a bit different, and..." The meeting falls into disarray, you overhear someone whisper to your boss, "Right, if I didn't have to follow the rules, I could be a hero too!" and the consensus finally emerges that it wasn't a fair race. Release is delayed; you are sent back in disgrace to "make things right."

Despite this setback, you eventually deploy, and senior management is impressed enough to try again. "This time, by the book," you mutter as you start your second Smalltalk project.

As the weeks go by, you notice that things aren't as smooth as before. Half the team isn't documenting to standards, and the other half are whining about having to get out of Smalltalk to run WordBlaster 6.0, or they're doing massive copy-and-paste documenting that is quickly out-of-date and never revisited. This time, you meet the corporate standards, but at a heavy toll in productivity.

Over 20 years ago, Donald Knuth had a similar problem. He noticed that software was not as "obvious" as the original author thought it was when writing it, even to the author himself after a few weeks! On the other hand, switching back and forth between coding and documenting was tedious and disruptive. So he invented "literate programming" in which the documentation is tightly bound to the code.

About the same time, Ted Nelson was dreaming about "hypertext," interconnecting all the information in the world in such a way that simply referring to something would take you deeper into its meaning.

As with many things, Knuth and Nelson were ahead of their time. The technology for literate programming was necessarily at a concrete level and batch-oriented, and hypertextual documents were more easily read than written.

A bit later, Adele Goldberg and a group of Xerox researchers were working on a programming system that, among other things, would greatly simplify both abstract expression and programming. The stage is set for hyper-literate programming!

PROCESS OF CONTINUOUS DOCUMENTATION

In a perfect world, there would be no programs. A computer user would describe a problem and a proposed solution in natural, but precise language, and feed it to the computer. This is a long way off—natural language is not precise enough, and computer language is not natural enough!

Keeping as close to that ideal as possible, we can set down some principles for documenting software "things"; a description of a thing must:

1. be on the same conceptual level as that thing
2. constantly and accurately describe that thing
3. be accessible; by creators, their peers, reusers, reviewers, end-user documentors, and the merely curious

Jan Steinman and Barbara Yates are cofounders of Bytesmiths, a technical services company that has been helping companies adopt Smalltalk since 1987. Between them, they have over 20 years Smalltalk experience. They can be reached at barbara.bytesmiths@acm.org or jan.bytesmiths@acm.org.

4. be measurable, both quantitatively and especially qualitatively.

Principle 1—Conceptual integrity

Driven largely by the limited abstraction available in traditional languages, most organizations have a limited bevy of documentation levels. These often follow the physical organization of the code: a function specification describes a single function, a module specification describes the functions in a file or directory, a system specification describes what you get when you type "make" or the result of some other build script.

Your documentation has conceptual integrity when it describes a software component at the same conceptual level as that component.

We specify Smalltalk software components at many levels, and add new ones as needed. Specifications we've found necessary are at the level of method, class, class extension, variable, nestable module, and configuration. (We'll explain each of these shortly.)

Every method must have a specification. Period. In fact, period at the end, capital at the beginning, and grammar throughout.

In addition, we've added other useful documentation components, such as gating checklists, requirements maintenance and tracing, and meeting minutes.

Principle 2—Constant accuracy

In the old days, it was simple. You opened both your ".c" file and your doc file in emacs or vi, and you worked on them simultaneously. Unfortunately, this only really works at a single conceptual level, thus violating Principle 1. Also, the popularity of WYSIWYG editors and specialized coding tools has weakened this binding, because many developers lack the memory and processing resources needed to have both their coding and documenting environments running at the same time!

We've adopted a simple strategy of which Knuth would probably approve: the documentation for a thing resides with the thing it describes. This has always been the case for methods, but is present in varying degrees for other components, depending on Smalltalk dialect and additional tools used.

At a more subtle level, we never "comment" our code, we "specify" it. A comment sounds optional, while even "cowboy coders" can appreciate the need for specifications—especially when they need them from someone else!

Principle 3—Accessibility

Much of the move to WYSIWYG tools for documentation has been driven by accessibility. A nicely formatted bit of paper can reach a much broader audience than can the programmer-accessible file `C:\PROJALPH\INPUTSYS-\MOD01334.DOC`, for example!

Simply adopting the Principle 2 tactic of co-residence for docs and software components vastly improves programmer accessibility, which makes continuous documentation practical, but this does little for the non-coding audience. Also needed is a way to "roll up" the conceptual levels in a hyper-access way that Nelson would find appealing. Although it should be discouraged, there will also remain the need to print a serialized version of an entire portion of the documentation tree.

Principle 4—Measurability

Traditional projects rarely measure project documentation, or they may only take gross physical line/file counts. What is needed is a qualitative measure that is significant in evaluating the overall project. In this sense, document reviews are much more important than code reviews, and are aided by compliance with the other three principles.

Simply having a project tollgate or milestone associated with documentation quality is not enough, however. To ensure compliance with Principle 2, it must be measured continuously. This does not mean daily, time-consuming review meetings; it means developing a team culture in which developers continuously refer to each other's latest documentation, and work together to correct inaccuracies on the spot.

Because of various documentation impediments noted above, the first thing you try when you want to use something in a traditional C project is often "grep the source." By having accurate, accessible documentation at the appropriate level of abstraction, the new ethic must be to first look at the documentation, and to immediately fix things if it is not what you need.

COMPONENT DOCUMENTATION NEEDS

Not surprisingly, different levels of abstraction have different documentation needs. Here's how we handle the different components.

Method specifications

Every method must have a specification. Period. In fact, period at the end, capital at the beginning, and grammar throughout—remember that any method specification might get "rolled up" into some serialized document that a VP will read! It only takes a moment—do the right thing.

As mentioned, we prefer the term "specification" instead of "comment." What does the method do? How are its arguments used? What objects are the arguments expected to be? What are the error conditions? What does the method answer?

As soon as you decide to create a method, capture in writing what you intend the method to do. (Of course, naming the method properly is vital, as Kent Beck has discussed in his column.) Developers often pay lip service to this rule, and in practice may only comment their meth-

ods when some process checkpoint demands that all methods have comments. It is much more difficult to comment a method after the fact, sometimes weeks or months after you wrote it. Or even worse, having to comment a method someone else wrote!

Put yourself in the shoes of a fellow team member who must take over the enhancement or maintenance of your code, or the member of a different team that is a client of your class. What should you tell the enhancer, client, or maintainer about this method so they can do their job well? (What should you write to keep clients from misusing your code and reporting "false" bugs against it!)

Tools that automatically generate accessing methods produce comments of little value. VisualWorks will generate instance variables and "getter" methods if you ask it to. The getter method comment simply states that it was auto-generated.

In general, "getter" and "setter" methods should specify the variable being gotten or set. What kind of object should go in here? Is it lazily initialized, and guaranteed to never be nil? How does changing it affect the containing object?

An organization needs a guide for its code and documentation.

We won't go into accessor method philosophy, except to say that they are not always appropriate. If you use tools that auto-generate such methods, realize that the tool cannot specify the meaning of those methods, and document them accordingly.

Class specifications

As soon as a developer decides to create a class, he or she must write a justification for the class. Why does the class exist; what does it do? Even during rapid prototyping, a minimum specification for the class is in order—it is a good work habit to have.

If you cannot yet describe the class at a high level, what sort of behavior are you about to implement for the class? The first specification you write for a class can be a rough draft, but it has to be there.

Throughout your further development of the class, you return to the class specification and add details, bring it up to date, and polish it. By the end of the current development cycle the specification will be accurate and complete. The spelling and grammar will be correct. As class owner, you should feel comfortable having your manager read it. (Or having your manager's manager read it!)

Class specifications are not written once and then forgotten. In each subsequent development cycle, the developer will review the specification and update it as required. The only time the specification is finished is when the class is no longer being changed.

Variables

Each class specification must have a section that documents all variables associated with the class: instance, class, class instance, pool variables, and (ouch!) globals. For each variable, its acceptable objects are listed, and a description is provided about how the variable is used. This is a good place to mention if the value is internally derived or can be set externally, and whether it is public or private. If the variable must be non-nil, the time and place where it is initialized should be spelled out. For example, if this is state that is provided at instance creation, point that out in the variable's comment.

We developed a technique for separating documentation of variables instead of embedding them in the class specification. This separates the specification of a class's state from that of its behavior, thus increasing conceptual integrity. It also allows superclass state documentation to be merged when printing or browsing.

Class extensions

Most Smalltalk code management systems distinguish between defining a class and adding behavior. Behavior is added in class extensions. Unfortunately, there is no built-in support for documenting class extensions. If you find it necessary to add a suite of methods to an existing class to support your work, shouldn't that need be explained somewhere? We find this to be necessary, and added class extension support to ENVY/Developer. The class extension comment summarizes the behavior added by the extension. Each method in the extension also has a complete specification.

Modules

Smalltalk source code management environments such as Team/V and ENVY/Developer contain software components that collect classes and/or class extensions. These components are called Packages in Team/V, and Applications or SubApplications in ENVY (which we'll simply call apps). The ability to support code modules larger than classes is essential in even moderate-sized projects! We're more familiar with ENVY, but the following discussion applies equally to Team/V.

We developed "smart" specification templates for apps that generate much of an app specification at the time the documentation is viewed or printed. This follows Principle 1 by letting the developer concentrate on subsystem documentation; Principle 2 by dynamically showing the state of included code modules as of the time of access; and Principle 3 by conditionally including class specifications and other detailed documentation chunks.

The most important part of the app specification is an abstract that explains its purpose and goals. The next section explains its component relationships. These two specification parts are maintained at this level; all other information is available via link, and maintained at a more appropriate place or automatically generated.

An important feature of the smart template approach is that organization-specific data is readily handled. We have sections for document control numbers, cost-center

information, and other data specifically required by the company. Paying attention to these things keep the "by-the-book" crowd happy, or at least tolerant!

Also in the app specification are links to automatically generated information, which is only feasible in a tightly integrated documentation system. For example, the app version and time stamp, prerequisites, class hierarchy, and system event method specifications are part of the automatically linked information.

Other information that we have included (via link) in the app specification are references, glossary, declared external interfaces, design decisions in the form of meeting minutes, requirements, use cases, test cases, and test results.

Linking information dynamically is more important in app specifications than elsewhere. The developer doesn't want the clutter of multiple "boilerplate" items that are not important to him, yet others may want to see everything. Good examples of linked information that is necessary, but should be hidden most of the time, are specifications of contained subapps and contained classes.

Configurations

ENVY and Team/V have ways of collecting modules into "load builds" of some kind. In ENVY, they are called *configuration maps*, or just *maps*. The specification for a map provides an overview of what it will load, and any other maps that should already be loaded. If for some reason the map is not unloadable, or is compatible with only certain versions of prerequisite maps, it needs to be documented.

An overview of what is different in one version of a map from the previous version is a good idea, but a better convention is generating and editing release notes at the end of each development cycle.

Release notes

ENVY and Team/V each have facilities for finding differences between two versions of components. We extended ENVY's facility to produce a smart template that captures all the changes in a textual form. Of course, ENVY isn't smart enough to say *why* something changed, but having a template to complete jogs the developer's memory, and guarantees coverage of all changes. We place these in the ENVY "notes" field of each changed application.

Diagrams

Documents without drawings are as unacceptable as mono-spaced, 80 column computer displays. Luckily, the publicly available HotDraw drawing framework is available. What it lacks in sophistication, it more than makes up for by being easily adapted to arbitrary object structures.

For example, the HotDraw diagramming inspector is suitable "out of the box" for documenting complex instance relationships that would be difficult to explain in words. Using it as an example, you can easily craft your own boxes-and-lines documenting aids.

We added a simple facility to ENVY for associating hot-drawings with arbitrary software components, linking those drawings with appropriate browsers, and embedding those drawings in hard copy at the appropriate place.

Style guide

Just as a magazine needs a consistent (or at least non-conflicting) style, an organization needs a guide for its code and documentation. Most Smalltalk projects start by searching for published style guides, adopting them, and modifying them as their needs evolve. We've found less attention is given to the style guide after the early stages of the project—typically, new hires are given the style guide to read. Hopefully, the developers have internalized the style guide, because they don't use it as a reference. It might come out of the bookcase again at code inspection time, or when the company is being audited for certification of the software development process.

Regardless of frequency of use, it is important to have one. There should be some agreement on what must be documented, and how it should be documented. Typically, style guides cover many areas in addition to documentation. Novice Smalltalkers should refer to the style guide, but, hopefully, they are also seeing good examples of documentation by fellow team members.

The need for the style guide is less important when there are good templates and tool support for documentation.

Documentation measuring

Beyond conformance to style and periodic peer review, the quality of documentation is difficult to measure. We've implemented existence checks, but they can't tell the difference between random characters and a line from Shakespeare. The best guarantor of quality is a group culture that encourages use.

Beyond mere existence checks, we've found a few tools that helped ensure quality documentation. Meeting minutes are linked into documentation to link important design decisions with the components impacted by those decisions, and smart checklists enable a developer to quickly assess their state of "doneness" for a given development cycle. Finally, we added a document-centered browser, so one could browse component specifications without being distracted by code.

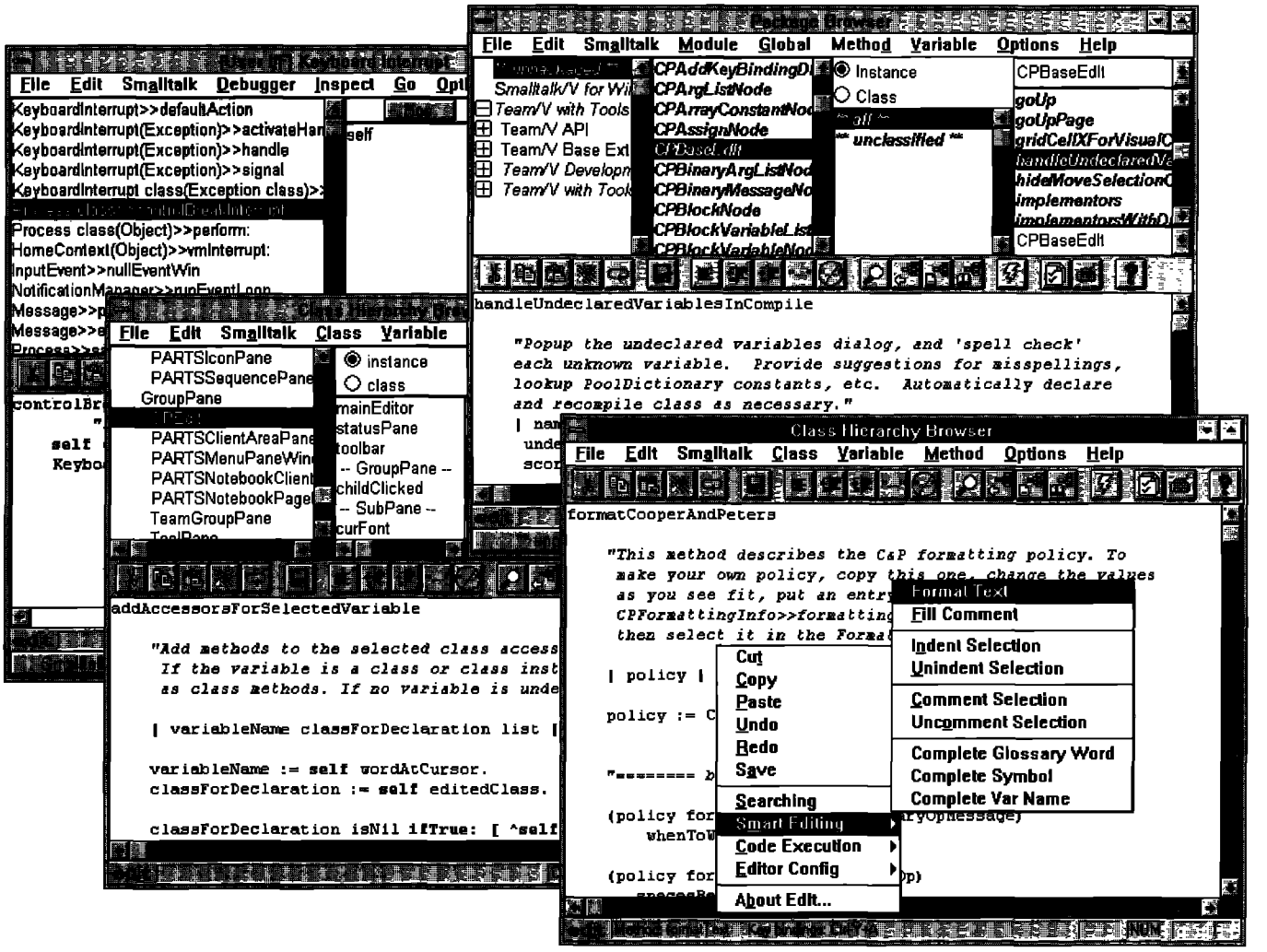
CONCLUSION

"Well, that all sounds great, but what do I do now?" Everything we've discussed here can be implemented fairly easily, depending on what is available for reuse in your environment. Although your schedule and resources may be such that "the cobbler's children have no shoes," it is also fairly easy to justify spending time building tool support for a continuous documentation regime. We've found it not only increased the quality of project documentation, but also resulted in a savings of about 8% of total project time, which means a team of seven people can justify a half-time toolsmith. In the next issue, we'll present some concrete examples and source code.

Cooper & Peters' edIt for Visual Smalltalk 3.0

Ron Charron

IF ONE WERE TO ASK A PROGRAMMER what kind of tool he uses the most, he would possibly answer "Oh, I'd say XYZ Smalltalk version X.Y." I guess we just take editors for granted. But take away a programmer's favorite editor, remove accelerator keys, or change it's behavior, and you had better stand well clear of the blast zone as he recognizes that someone messed up his image. Because Smalltalk development environments all come fully equipped with syntax-checking editors, Smalltalk programmers seldom (if ever) resort to an external editor. If moving to Smalltalk from another language, you'll probably complain for a while, then lose your complaints as you discover a nicely integrated environment provided by your Smalltalk. So, what if you are finally offered the option to use a better editor for your Smalltalk environment? If you're like me, you'd probably say "why bother?" But, after some 10 years or so using Smalltalk, I do believe that Cooper & Peter's edIt is the first commercially available add-on editor



edIt for Visual Smalltalk 3.0

We'd like to hear from you...

if you'd like to play a significant role in a large object-oriented development project to deploy business information systems throughout an enterprise. OOCL's IRIS-2 project takes a strong software architecture approach to building an integrated information infrastructure. By creating an extensible architecture, we are better positioned to accommodate future changes in the business. VisualWorks/ Smalltalk is the development platform.

The IRIS-2 development team is based in Santa Clara, CA. OOCL, an industry leader in the containerized shipping business with over 140 offices around the world and 2000 employees, offers reliable transportation services to its customers via a global network of ocean and intermodal routes.

Project Manager

Reporting to IRIS-2 senior management, you will manage a group of Software Developers and be responsible for iterative and incremental development and delivery. Your 5+ years of management experience must reflect strong leadership and people skills, team building, working with changing priorities, and a track record of managing projects to on time, on budget delivery. Technical hands-on experience with OO and Smalltalk development is desirable.

Smalltalk Developers

We are looking for experienced VisualWorks/Smalltalk developers with strong interest in domain modeling, user interface design, and persistence and distribution technologies. You will have the opportunity to work with a highly skilled, highly motivated Smalltalk development team in an environment which emphasizes technical excellence, teamwork and professional growth. If you are OO fluent and eager to join the league of the very best in Smalltalk development, we'd like to talk to you.

Productivity Tools and Release Engineer

We are building a team to provide the OO tools and infrastructure for software delivery. If you have experience in configuration management, release engineering, and tools and utilities development, you can play a role in helping us build quality into our development process.

OOCL offers competitive compensation packages and the technical and analytical challenges you expect in a state-of-the-art environment. Apply by sending your resume to Lori Motko via e-mail, indicating the position of interest, at motkolo@oocl.com, or mail to OOCL, 2860 San Tomas Expwy, Santa Clara, CA 95051, or fax to (408) 654-8196.

OOCL

Dedicated to Quality Service

Recruitment Center

SMALLTALK POSITIONS

DIGITALK is seeking experienced Smalltalk instructors and consultants for our world-class Professional Services team. At **DIGITALK** you will work with one of the world's leading development teams, use state-of-the-art products and assist companies on the forefront of adopting object technology in client-server applications.

Requirements for Senior Consultants are: solid experience with Smalltalk (3-5 years) and/or PARTS Workbench experience. OOA/D experience and GUI design skills. Mainframe database experience is a big plus. Requirements for instructors are: previous training experience in a related field (2-4 years), understanding of OO concepts and Smalltalk.

Positions are available in various sites throughout the U.S. Compensation includes competitive salary, bonuses, equity participation, 401(k) and family medical coverage. All positions require travel. **DIGITALK** is an equal opportunity employer.

Please forward your resume to:
Director of Enterprise Services
Digitalk, Inc.
7585 S.W. Mohawk Drive
Tualatin, OR 97062
fax: (503) 691-2742
internet: holly@digitalk.com

DIGITALK

ObjectSpace

Object Technology Professionals

ObjectSpace, Inc. is a cutting-edge leader in the object-oriented arena with awesome technological capability and extraordinarily talented people dedicated to the creation and deployment of advanced technologies.

Progressive growth has created immediate career opportunities for Object Technologists who are highly technical and are committed to excellence.

We have requirements for Object Technologists who have strong object-oriented backgrounds and two years of experience in one or more of the following:

Smalltalk	*Distributed Smalltalk*
C++	*VisualWorks*
Fusion	*VisualAge*
Rumbaugh	*Booch*

We offer competitive compensation, performance-based and travel bonuses and a complete benefits package. For consideration, send a resume to:

ObjectSpace, Inc.
14881 Quorum Drive, Suite 400
Dallas, Texas 75240
1-800-OBJECT1
Fax: (214) 663-3959
jobs@objectspace.com

for Digitalk Smalltalk. And because Cooper & Peters were the team that originally introduced such helpful tools as WindowBuilder, their editor was certainly worth giving a try.

WHAT IS IT?

EdIt is a replacement for VisualSmalltalk's base editor. All basic functionality is still there, but you will find many improvements. In addition to the options you would regularly find in the base editor, you will find "Searching" and "Smart Editing" options off the editor's pop-up menu. Many options are also available through a configurable editor toolbar.

Syntax highlighting

Being in the Smalltalk training business, I am constantly needing to highlight a Smalltalk student's mistakes and help correct them. Because edIt's syntax highlighting helps better distinguish the various elements in the code, anyone making his first steps in Smalltalk would probably find this feature useful. Constants, comments, and keywords are highlighted through the use of colors, italics, and bolding. EdIt easily allows you to reconfigure highlighting to suit your individual preferences.

Assisted variable declaration

When saving a method in a class browser, people using ParcPlace's VisualWorks have long been used to having a menu pop up when the method contains unknown references to instance variables or globals. They are offered a choice of declaring the unknown identifier as an instance variable or a global, etc. EdIt finally brings this feature to the Digitalk world. I must admit that I have often found that feature annoying while coding in VisualWorks. I find myself saying "yeah I know, I know, I'm going to declare them (instance variables) all at once a bit later—Stop nagging" But on some occasions, I must also admit that I've found the feature useful. In a way, this feature can lead you to get a little lazy, by getting used to not declaring instance variables as a formal coding step. Use what you want while coding a method, and then when you save edIt will figure it out for you, and present you with variable declaration options. You will be offered the choice to declare your variables as temporary, instance, global, class, class instance, or, in some cases, to have edIt set up a pool dictionary containing a reference to a known global. If edIt can recognize enough of your misspelled identifier, it will offer a replacement suggestions list, just like a spell checker utility.

Variable and text completion

EdIt provides a user-maintainable glossary used for text expansions. Type in a few characters, invoke an expansion

command, and edIt will expand the word to the closest match in the glossary. You can also expand from instance variable names and globals. Although by default you need to pop up a menu to invoke the expand commands, you would either want to assign a keystroke to invoke them, or customize the toolbar to make this truly useful.

Searching facilities

EdIt offers a bounty of searching options, some that can be invoked very conveniently. For example, in a method editor, position the text cursor on a method selector for any message send and call up the "smart-editing menu"; you can easily browse the implementors or the senders. EdIt also offers "qualified" senders and implementors options that allow you to specify the scope of the search (whole image, or some level within the inheritance hierarchy for your class). Regular expression searching is supported. A scoped search and replace facility is also included and can also go across modules if you're using the module manager.

Other nice features

EdIt provides a "bottomless" undo and a redo feature for those days when you find yourself uttering "oops" a little too often. Also, edIt's key binding facility will delight people who find that a mouse just gets in the way of getting things done. Any editor function can be bound to keystrokes. C & P have even given edIt the capability to bind keystrokes to your own methods and save your key bindings in key sets. Some of you may be delighted to hear that Epsilon and Brief key bindings are included with edIt.

CONCLUSION

C & P have put some effort into making edIt extensible. Source code is included, and the help facility is very good (it's nice to see that help screens have finally made their way into the Smalltalk industry). If you're like me, and like snooping around under the hood, you will find a few hidden goodies in the supporting C & P class library. There is a change set manager lurking in there, but I didn't try adapting it for general use.

EdIt is available now for Visual Smalltalk 3.0 Win32, and the OS/2 version should be available in late June, 1995. The list price is $195 (Win 32), and a fully functional demo is available upon request (call 303.546.6828).

I've been using C & P's edIt for a few months now, so I've had a chance to let the "newness" aspect dissolve a bit. EdIt is a tool that can grow on you. Use it for a while and then try to take away its features, and you will find yourself looking for them. But then, I had made that point about editors at the beginning of the review, didn't I?

Ron Charron is Director of Corporate Services at The Object People Inc., Ottawa, ON, Canada. He spends most of his time "immersing" corporate developers worldwide into the primordial Smalltalk soup. He can be reached by email at ron@objectPeople.on.ca or, for longer periods of time, through an Immersion Program, v: 613.225.8812.

Product Announcements

Product Announcements are not reviews. They are abstracted from press releases provided by vendors, and no endorsement is implied.

Vendors interested in being included in this feature should send press releases to THE SMALLTALK REPORT, Product Announcements Dept., 885 Meadowlands Drive #509, Ottawa, ON K2C 3N2, Canada, 613.225.8812 (v), 613.225.5943 (f).

VISUALWORKS SUPPORTS POWER MACINTOSH

ParcPlace Systems Inc. announced the availability of its VisualWorks client and server application development tool for Apple's RISC-based family of Power Macintosh computers. VisualWorks for the Power Mac is a native application optimized to take advantage of the computer's power. Applications written in VisualWorks are instantly portable across all major client/server platforms, including: Windows, Windows NT, OS/2, Macintosh, Power Macintosh, and major UNIX-based systems. Through VisualWorks' dynamic compilation, which compiles source code to the computer's native instruction set when needed, developers need only develop their code once. The finished application can be deployed across all platforms without any recompiling or reprogramming effort. In addition, VisualWorks' cross-platform portability ensures that capabilities usually available on one system, such as a notebook or combo box, can be extended to all supported platforms. This allows developers to concentrate on building applications rather than learning different windowing systems.

ParcPlace Systems Inc., 408.720.7514.

DOCUMENTATION AND REUSE TOOL FOR IBM SMALLTALK

Synopsis Software, a provider of object-oriented development tools, released Synopsis for Smalltalk for IBM Smalltalk. The automatic documentation of classes is an important factor in producing reusable components in Smalltalk. Synopsis is an automatic class documentation tool for development teams using IBM Smalltalk. Synopsis also allows developers to print their class documentation with popular word processors, eliminating the time-consuming task of converting plain text from the Smalltalk environment into word processor documents.

Synopsis produces documentation summaries of individual classes; builds class encyclopedias, in which many class summaries are gathered together in the form of an interactive class reference manual; exports documentation summaries to popular word processors; packages documentation as encyclopedia or Help files; produces source code listings for classes; and supports personalized documentation and coding conventions. Synopsis is available for both the Team and Standard versions of IBM Smalltalk. Windows and OS/2 platforms are supported.

Synopsis Software, 919.847.2221