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)

Table of Contents

May 1995 Vol 4 No 7

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

Well, we're pleased to report to you that the inaugural Smalltalk Solutions conference went off without a hitch (with the exception of the hotel itself!). The conference was very well attended, with an interesting mix of experienced hard-line Smalltalkers and novices wanting to find out more about the technology. The feedback we received was that the hard-liners found the get-together to be a very informative week, offering the chance to explore new uses for Smalltalk and discover how others are making use of the technology. Many of the novices with whom we spoke were "very intrigued" with the exuberance of the people attending. They seemed to sense that the culture surrounding Smalltalk is different from other software engineering communities. And as this community continues to grow at such a rapid pace, it's interesting to see that it hasn't lost this collegial aspect. From our point of view, we found the conference to be an excellent opportunity to meet old friends and speak to many of you. So for the two dozen or so of you who suggested that you'd like to contribute to THE REPORT, let's hear from you <grin>! Be sure to mark next year's date (and new hotel) on your calendar—it's March 4–7 at the Marriott Marquis in New York.

While Smalltalk proves to be useful for exploring a design, it ultimately is just code

One of the interesting topics being discussed by many at the conference was what documentation is necessary for designing Smalltalk systems to be constructed, as well as what documentation must be generated to explain the system once it is delivered. We encounter this question regularly in our dealings with clients, and it proves to be a difficult one to answer. Many projects we know of are generally not following any of the Booch, Rumbaugh, or Shlaer/Mellor methodologies, or at least not to their full extent. In fact, it's not clear to us that the techniques used by these approaches are necessarily a great fit to Smalltalk development. One interesting thing about Smalltalk is its suitability as a language for expressing and evolving a *design* as well as an implementation. It's not that the notation put forward by the methodologies listed above isn't useful, it's just that it is often more than necessary. Mastering the notations associated with these approaches is a very expensive activity, and what you are left with is still a "paper" design. Many people find that applying a mix of Responsibility-Driven Design along with Use Case analysis is sufficient to deliver with their projects.

What we have seen being done by many groups is to work out a design, but not necessarily document it using a formal notation. Instead, they work out their design as a rough sketch, which they simply take away and construct in Smalltalk. This often leads to highly successful projects; particularly from the point of view of putting systems together quickly and effectively. The downside to this approach is that without documentation describing the design and architecture, we're still building legacy systems that will suffer in the long run. The problem is that, while Smalltalk proves to be useful for exploring a design, it ultimately is just code; it is often difficult for people who were not directly involved in the creation of a system to extract the design of an existing implementation. The true test for any design architecture comes when a new group takes over the maintenance of a system, and the original team working on it is no longer available!

There are different approaches to address this problem. Number one in our minds is to put in place an architecture that is as simple and elegant as possible, while isolating—as much as possible—the things that aren't simple from the mainstream developers. The idea is to make it as straightforward as possible for the "average" maintenance developer to do his/her job. This sounds like an oversimplification of the problem, but in virtually all systems this separation of complex aspects of the architecture is achievable with just a little extra effort. Second, make sure their are lots of examples of how the system is intended to be used. We write example methods, which are class methods for each class created. These methods illustrate the different ways to interface with our classes. It's unfortunate that the vendors don't do this as well, for they must have these examples for their own internal use. Third, it is important during design for the group to set standards for building their system, and that these be adhered to strictly. This includes issues such as choosing names for messages that clearly indicate their intent; being careful to choose either singular or plural names for parts and messages; making sure similar messages are consistent with their return types; and making sure that arguments to messages are not be constrained by the class of object being passed, but instead rely strictly on the behavior of the argument. These are all little things that take an extra few minutes to check when you're writing the class that will save weeks of effort in the future.

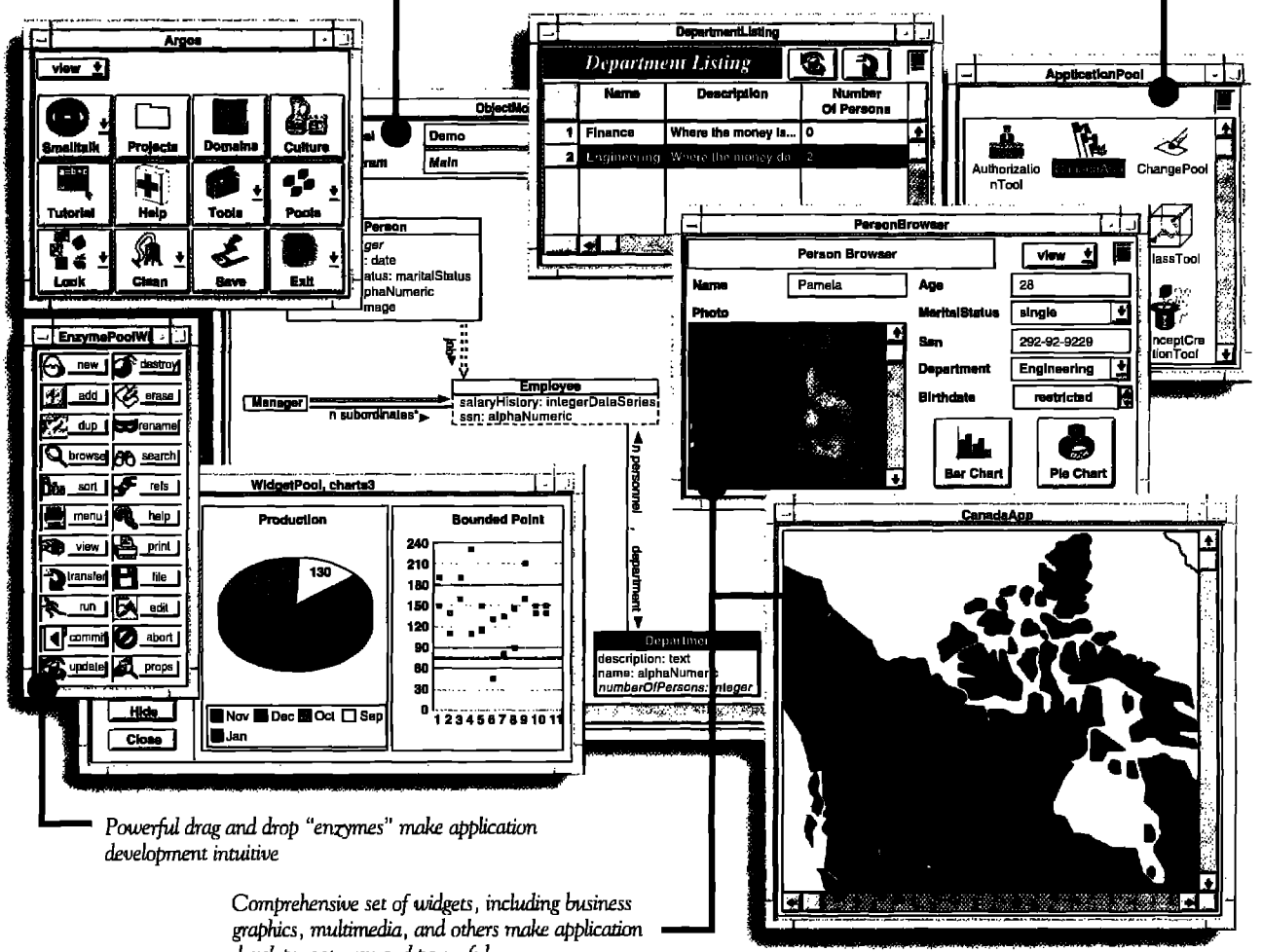
Enjoy the issue.

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

An O-O approach to accessing external resources

Yoel Newman & Michael Parvin

IN THE COURSE OF OUR DEVELOPMENT, we have extended the VisualWorks development environment by allowing access to external resources such as communication protocols, database access, and multimedia services. This article will discuss the approach we have taken in incorporating these features into VisualWorks 2.0. We will also give a brief overview of VisualWorks' often overlooked DLL and C Connect (DLLCC) product (referred to as C Programmers Object Kit (CPOK) in VisualWorks 1.0).

DLL AND C CONNECT: A SIMPLE EXAMPLE

When your Smalltalk application has to communicate with an external component, you will need to use a function library. The function library will also provide an application programming interface (API). The API typically contains declarations that specify the function prototypes implemented by the library. In the OS/2 environment, the function library will most probably be released as a dynamic link library (DLL) because DLLs reduce program size and can be shared by more than one program. We will start with a short example showing how to use DLLCC to make an API call from within Smalltalk. Although DLLCC has many other capabilities, we will only demonstrate bringing a DLL into a Smalltalk application. Remember that this example is intended merely as a quick overview. For more complete and extensive coverage, please refer to the DLLCC manual.

DLLCC introduces a new abstract class called ExternalInterface. Accessing C functions requires subclassing this ExternalInterface class. DLLCC also extends the Smalltalk syntax to allow C language declarations within Smalltalk methods. This extended syntax is only available to subclasses of ExternalInterface. A sample declaration could look like this:

```
RETCODE
    <C: typedef integer RETCODE>
or like this:
SQLAllocEnv: handle
    <C: integer SQLAllocEnv(unsigned long * handle)>.
```

To create an ExternalInterface subclass, you will need access to both the DLL files and the header files (usually .h files). With these files, you can create an ExternalInterface subclass either by manually entering the methods that correspond to the C declarations, or by having DLLCC parse the C header files and automatically generate a method for each C declaration. There is also a Builder tool that allows you to examine header files and selectively generate the ExternalInterface subclass.

In our example, we will call the DosBeep function from within the OS2 library. We will be using OS/2's doscall1.dll. The API is defined in the file bsedos.h. The first step is to define the ExternalInterface subclass in the System Browser.

```
ExternalInterface subclass: #DosCalls
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'STR'
```

This results in the following new template:

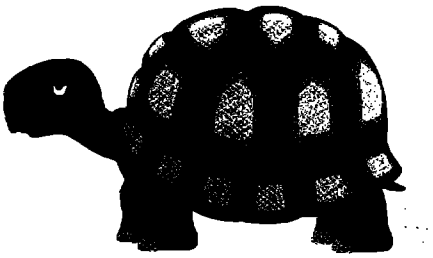
```
ExternalInterface subclass: #DosCalls
    includeFiles: ''
    includeDirectories: ''
    libraryFiles: ''
    libraryDirectories: ''
    generateMethods: ''
    beVirtual: false
    optimizationLevel: #debug
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'DosCallsDictionary '
    category: 'STR'.
```

Then fill in the fields as follows:

```
ExternalInterface subclass: #DosCalls
    includeFiles: ''
    includeDirectories: ''
    libraryFiles: 'doscall1.dll'
    libraryDirectories: 'e:\os2\dll'
    generateMethods: ''
    beVirtual: false
    optimizationLevel: #debug
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'DosCallsDictionary '
    category: 'STR'.
```

The libraryDirectories and libraryFiles refer to the DLLs to be loaded. Make sure to fill in the correct drive where the OS/2 system is installed. The includeDirectories and includeFiles refer to the header files to be parsed. If there are includeFiles specified and there is a "*" in the generateMethods field, then the header files will be parsed. Since we are only call-

PICTURE THIS ...



THIS COULD BE YOUR OBJECT ORIENTED PROJECT

THIS COULD BE YOUR OBJECT ORIENTED PROJECT WITH TRS

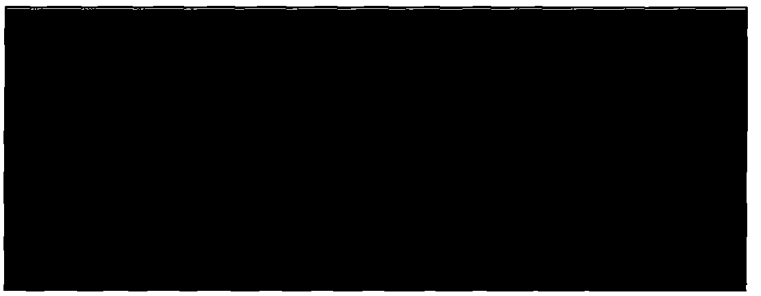
ANY QUESTIONS???

- Consulting
- Mentoring
- Application Frameworks
- Methodologies
- Training / Immersion Programs
- Reusable Components
- Project Reviews
- Coding

"Where Smalltalk Talks Big"

TRS

TECHNICAL
RESOURCE
SOLUTIONS



3900 W. Alameda Ave., Suite 1700, Burbank, CA 91505, 818-972-1744, 800-801-1TRS (877), Fax 818-972-1685
WWW: http://www.primenet.com/~trsinet email: trsinet@primenet.com BBS: 818-972-1642 - 14400,8,N,1

ing one function, we don't want to parse the entire header file. We will manually add the following method on the instance side:

```
DosBeep: freq with: duration
    <C: unsigned long DosBeep(unsigned long freq,
unsigned long duration)>
    ^self externalAccessFailed
```

Under most circumstances we would be done here. However, in the case of the doscall1.dll there is one final step. This is because doscall1.dll does not export all its functions by name, so they must be referred to by ordinal rather than name. Therefore, we must resolve the ordinal before we can call it from our application. There are two ways we have found to do this. The first way requires that you determine the ordinal of the function you need using exemap, exehdr, or some local guru. In our case, DosBeep is ordinal number 286. With the function ordinal , you can execute: (DosCallsDictionary at:#DosBeep) ordinal:286. This will cause DLLCC to resolve the function by ordinal and not try to resolve it by name. (This handy bit of information is in the manual under MS-Windows 3.1 platform-specific information, but we have found that it works just as well in OS/2).

The second way is a bit more complicated and may be more familiar to C programmers. You will create your own version of the DLL that forwards the request to the actual OS2 DLL.There are four steps.

1. Create a "dummy" function that looks like this in foo.c:

```
int foo()
{}
```

2. Create a DLL definition file mycall.def:

```
LIBRARY MYCALL
PRTMODE
EXPORTS
    DosBeep
```

3. Compile to object code only as follows (using IBM CSet Compiler)

```
icc /c foo.c
```

4. Link to the OS2 library to create mycall.dll:

```
link386 foo.obj, mycall.dll,, \ibmcpp\lib\dde4sbs.lib,
mycal.def /NOI
```

and use mycall.dll in the class template for the libraryFile parameter instead of doscall1.dll.

To use the function, execute:

```
DosCalls new DosBeep:1000 with:1000.
```

DLLCC is a powerful product and will have no problem parsing ANSI-C header files. Be warned that there may be work necessary to get your files to parse correctly. There are, for instance, many cases where the header files must be tinkered with to parse correctly. Also, the parser ignores some compiler directives like #pragma and #line.

THE OBJECT PEOPLE

Training from the Experts

SUMMER/FALL 1995
Open Course Schedule

INTRODUCTION TO VISUALAGE
JUNE 12–16
JULY 10–14
AUGUST 7–11
SEPTEMBER 11–15
OCTOBER 9–13
NOVEMBER 6–10
DECEMBER 4–8

BUILDING APPLICATIONS WITH VISUALAGE AND IBM SMALLTALK
JUNE 19–23
JULY 17–21
AUGUST 14–18
SEPTEMBER 18–22
OCTOBER 16–20
NOVEMBER 13–17
DECEMBER 11–15

PROGRAMMING IN VISUAL SMALLTALK
JUNE 5–8
AUGUST 21–24
SEPTEMBER 25–28
OCTOBER 23–26

PROGRAMMING IN VISUALWORKS
JUNE 26–29
AUGUST 28–31
OCTOBER 2–5
NOVEMBER 20–23

OO CONCEPTS, ANALYSIS & DESIGN
JUNE 5–7
JULY 31–AUGUST 2
SEPTEMBER, 6–8
OCTOBER 10–12
NOVEMBER 1–3

PROGRAMMING IN IBM SMALLTALK
JUNE 5–8
AUGUST 21–24
SEPTEMBER 25–28
OCTOBER 23–26

VISUALAGE FOR SMALLTALK PROGRAMMERS
JUNE 26–29
AUGUST 28–31
OCTOBER 2–5
NOVEMBER 20–23

The courses are presented in Ottawa, Ontario, Raleigh, NC, and Southampton, England

THESE COURSES ARE ALSO OFFERED AT YOUR SITE. CALL FOR DETAILS.

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

DO WE WANT TO USE AN ExternalInterface SUBCLASS?

Once you have generated an ExternalInterface subclass, you have a class with several methods. Each method has the ability to call one of the functions contained in your DLL. But how should you call these functions? The easiest approach might seem to be just calling a function directly , through its corresponding method, whenever you needed to use one. There would, however, be several disadvantages to this approach.

To scatter function calls everywhere would make the application difficult to understand. It would leave no clear indication of what a function is being used for and how it is being used. Also, it wrongly assumes that everyone working on the application (now and forever after) knows how to use the function library. We want a way to encapsulate the API's behavior in a way that makes it clear how to use it and how it is being used.

There will also be many times when you will need to find all your function calls. The vendor might release a new version of the library, causing you to change your code, or you might decide to go with a different vendor. You might also need to do some system maintenance or extend the system. In all these instances, you want to touch as few areas of your system as possible. This will be hard if you have your function calls in many parts of the system. We need a way to minimize the impact of system changes.

Calling the function calls directly also breaks portability. For example, if we have hard-coded calls that open the OS/2 file dialog window then we can no longer run our application unchanged on any other platform. It would be nice to use the same call to bring up an OS/2 file dialog under OS/2 and a Windows file dialog under Windows.

Finally, raw function calls within Smalltalk code will result in ugly and non-intuitive code that tends to look more like C than Smalltalk and will seem unnatural to the Smalltalk programmer. Imagine a code snippet that looked something like this:

```
interfaceClass := SomeExternalClass new.
retCode := interfaceClass call:parameter1 with:parameter2.
retCode == interfaceClass GOOD_RETURN_CODE
    ifTrue:[ self doSomeCode]
    ifFalse:[DosCalls new dosBeep:parameter with:100]
```

What we want is a way for Smalltalk programmers to use the functions in the same way they have learned to use the rest of the system. This is especially true if the people using the functions are not the same people who are bringing them into Smalltalk.

All these problems leave us with a design dilemma. What is the best way to use a function library in a Smalltalk system?

ELEMENTS OF A HOST PLATFORM ACCESS FRAMEWORK: VisualWorks

Creating a host platform access framework is an excellent way to solve our dilemma. This can be can be done using a multiple layer approach (see Fig. 1).

The API access layer is the lowest layer in the framework and provides access to the API's functionality. The API access layer is a subclass of ExternalInterface. This subclass contains the procedures, structures, typedefs, and manifest constants for the API as well as the behavior necessary to call the functions in a DLL from the Smalltalk environment. This layer contains only the essential code required to make the function call.

The API wrapper layer provides a higher-level interface for the API functionality. Conceptually, it is the next layer in the host platform access framework. The layer is composed of an abstract class and its concrete subclass implementors. The API wrapper layer wraps API function calls and

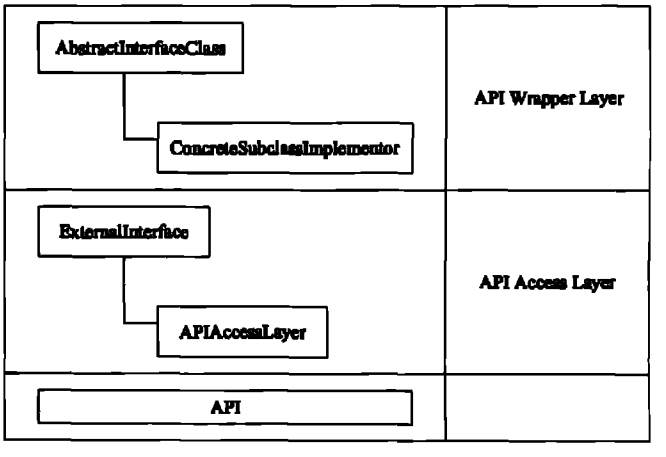


Figure 1. Object model relationships.

shields the Smalltalk application programmer from the implementation details of the API itself. The separation of the interface from the implementation allows the development of the API wrapper layer to proceed iteratively.

The abstract class defines a generic interface for the behavior exhibited by the "family" of APIs to be supported in the Smalltalk environment. This "family" can range from a single API, to a group of vendor-specific database APIs, to a group of APIs that perform the same behavior for specific platforms. The abstract class's design should be robust enough to support additional implementations for other platforms or from other vendors.

Concrete subclasses implement the specifics for each host platform or vendor-specific offering. The portability of API functionality is possible because the subclass implements the operations to support the abstract interface. The advantage of using an abstract class to define the interface and concrete classes to support the implementation is that users of the framework only commit to the interface defined by the abstract class, not to a particular implementation defined by a concrete class.

What is the best way to use a function library in a Smalltalk system?

The concrete subclass for an abstract API wrapper class will have to handle the following items in its implementation.

- Memory allocation and deallocation.
- Structure allocation, deallocation and member accessing.
- Exception handling.
- Maintaining and enforcing the state of the API.

MEMORY ALLOCATION AND DEALLOCATION

The concrete implementors will handle the allocation and deallocation of memory on the external heap. Typically, API calls will require parameter passing. In Smalltalk, this means that the API parameters need to reside in memory explicitly allocated on the external heap, i.e., memory that is not managed by the Smalltalk memory manager. Initially, the internal implementation of a concrete subclass could use gcMalloc and gcCopyToHeap to allocate memory on the external heap. The "gc" refers to the garbage collectible nature of the memory pointers on the external heap. When a garbage collectible pointer is no longer referenced, its finalization mechanism frees the memory on the external heap. This finalization mechanism takes care of any implementation memory leaks associated with allocating and deallocating memory. Later in the development process, more advanced memory allocation techniques could be used such as allocating and deallocating the memory explicitly for more performance. Even though gcMalloc and gcCopyToHeap facilitate the development of the API wrapper layer by providing automatic allocation and deallocation of memory, they do so at a high performance cost. Explicitly allocating and deallocating memory will increase the overall performance of the framwork by removing the overhead of the garbage collectible pointers finalization mechanism.

Memory allocation technique.

```
[ "Begin unwind block"
 ptr1 := CIntegerType unsignedLong malloc.
 ptr2 := CIntegerType unsignedChar malloc.

 ...
 ...
 ...
 "End unwind block" ] valueNowOrOnUnwindDo:
    [ ptr1 notNil ifTrue: [ptr1 freePointer ].
    ptr2 notNil ifTrue: [ptr2 freePointer ] ].
```

The use of valueNowOrOnUnwindDo: to handle the API call is an example of a memory allocation technique for explicitly allocating and deallocating memory on the external heap. The handler code will be evaluated whether or not an exception occurs. An interesting aspect of the code is that the message freePointer is being sent to the CPointer rather than free. The freePointer method is faster than free if it is known the pointer is not garbage collectable.

Another performance item to take into consideration in the internal implementation is the allocation and deallocation of large memory regions. The environment is incurring a performance penalty with every function call with no discernible gain. Instead, the CPointer could be stored in a concrete subclass instance variable and reused in place of memory pointers with a local scope.

A key point to mention here is that the actual implementation details of memory allocation and deallocation remain hidden and separate from the interface. Tweaking for performance and style can take place later in the development process without breaking the interface, and should not be the primary concern when initially developing the implementation.

STRUCTURE ALLOCATION, DEALLOCATION, AND MEMBER ACCESSING

Structures are one type of parameter passed in an API function call. From the previous discussion, this implies that a structure will need to be allocated on the external heap. The concrete subclass is responsible for allocating the structure, filling in the necessary members, and retrieving the data from the members after the function call returns.

In Smalltalk, we do not want to require the application programmer to allocate a structure, fill in the members, and deallocate the structure when it is no longer needed. For instance, funcX, an API function, requires as a parameter a structure containing three members. To use funcX an application programmer would require implementation-specific knowledge to allocate the correct structure and fill

in the correct members to satisfy the requirements of the function call. In contrast, a more object-oriented interface would provide a keyword message taking as arguments any additional information not already known to the implementation. The concrete implementation would convert this information into the format required to make the function call. The keyword message is specified by the abstract class and is only implemented by the concrete subclass. A different concrete subclass may not convert the information into a structure but rather may pass each parameter to a function call. The differing implementation details would have no impact on the user of the API wrapper layer because the interface for the concrete subclass implementors is always the same. In addition to separating the API functionality from its implementation, the abstract class defines an interface that provides more insight into the behavior performed by the concrete implementors.

EXCEPTION HANDLING

The concrete implementors not only have to handle exceptions when they occur, but must also provide a hierarchy of exception handling signals to resolve any API call failures. These signals will provide a portable way of handling exceptions. For example, an exception handling hierarchy for a communications interface would contain at the highest level a communicationsErrorSignal, and then lower in the hierarchy a connectionErrorSignal and a stateErrorSignal. Each subclass implementor will generate these exceptions for a given set of error criteria. As with the design of the abstract interface, the exception handling hierarchy needs to be designed as a set of generic exceptions that support the common functionality contained within the family of APIs. Even if this "family" is made up of a single API library, specific references to platform, vendor, and product should be avoided, e.g., an exception called databaseErrorSignal would be a better design choice than an exception entitled db2ErrorSignal. As an example, the VisualWorks External Database Interface, which will be discussed later, includes the following exception hierarchy:

```
externalDatabaseSignal
    connectionExceptionSignal
        authenticationFailureSignal
        connectionNotOpenSignal
        unableToConnectToSQLserverSignal
            unableToConnectToSQLenvironmentSignal
```

The exception handling hierarchy provides the primary source of error detection information to the client. It should be the only source of exception information available to the client of the API wrapper layer. The failure information returned by the API should only be used in the internal implementation of the API wrapper layer. The exception handling hierarchy can be enhanced with information returned by the API; however, users of the framework should not rely on API-specific error information in their implementation. The specific API error information should only be used as a troubleshooting tool during development. More importantly, reliance on API-specific error code information in a client implementation will break portability across multiple platforms and across different vendor implementations.

Maintaining and enforcing the state of the API

The API wrapper layer has a responsibility to maintain a representation of the state of the API. This is only necessary if the API functionality is dependent on its current state. For instance, in a hypothetical communications protocol, the flow of communication verbs needs to be Allocate, Send, Receive, check Receive for more data flag, if it exists then Receive, if it doesn't then OK to Send,... , Reset. This hypothetical protocol is highly dependent on the state of the session because issuing a Send verb before an Allocate verb would cause a communications exception. On the other hand, an API that only contains functions that display various file dialogs would have little need for the concept of state. However, for those APIs that rely on state to dictate the action taken during a function call, the API wrapper layer needs to maintain an accurate representation of its state, perform state transitions when necessary, and raise an exception when an attempt to call a function that does not support the current state is executed.

The portability of API functionality is possible since the subclass implements the operations to support the abstract interface.

The state of the API can be maintained using a concrete subclass instance variable storing a symbol representing the current state. The concrete subclass would use case-style statements to check the state before executing a function call. Another approach would be to create a State class that not only maintains state but can be instantiated and initialized with a lookup table that specifies the messages that can be sent for a specific state. The API wrapper layer will still be responsible for making the state transitions, but the proliferation of case-style statements can be avoided.

FRAMEWORK CONSIDERATIONS

With the framework approach, we have solved the problems mentioned above. We have a common interface that can be called from anywhere in the application. Programmers can use the framework knowing only the interface. They do not need to know the API itself because we are masking the implementation details. With this arrangement, users are committing only to the implementation and not to the platform. Separating the interface from the implementation will allow us to subsequently change platforms without breaking the code.

We have also localized all access to external resources.

We are only calling the functions directly in the API wrapper layer, which masks the actual implementation. This makes maintenance and upgrades easier because we can change the implementation without affecting the users of the interface. Also, because we will be allocating space on the external heap, we can keep track of these allocations to avoid problems such as memory leaks.

There are some things to keep in mind when creating an external access framework. Users will be tied to the implementor's view of what the interface should be. The framework needs to define an interface for all implementations, but it can be hard to create an abstract interface that can account for all the differences across implementations. It is possible that the framework implementors only know one platform. Also, there will be situations where some features are only available on certain platforms and a decision needs to be made whether to exclude these extensions in the framework and keep the interface uniform or to extend the implementation class and give it a larger interface. Finally, the framework needs to encompass future products as they become available. It is important to commit some time and effort to creating a truly robust framework.

THE EXTERNAL DATABASE INTERFACE EXAMPLE

An excellent example demonstrating the above approach is found within the VisualWorks system itself. VisualWorks needed the ability to communicate with several database management systems. However, each DBMS is different and each DBMS requires the use of a separate API. We will look at how database connectivity is implemented in VisualWorks to see how it fits our model.

At the highest layer we have an abstract implementation to which we should program. This layer is included in every VisualWorks system and is contained in the External Database Interface (EXDI) classes. These classes are found in the category "Database-Interface" and include such classes as ExternalDatabaseSession and ExternalDatabaseConnection. These are abstract classes representing a framework for external relational database access. All behavior necessary to interact with a database (such as connecting, disconnecting, executing an SQL statement, or initiating a transaction) is defined in these classes and documented in Chapter 13 of the user's guide. These classes provide the framework for all database access but do not provide access to any database in particular. There is also a complete error handling hierarchy defined. The framework relies on concrete subclasses to provide the implementation details specific to each database platform. The concrete classes are provided for both Oracle and Sybase and are packaged as separate Database Connect products. If you have these products installed, then the classes can be found in the "Database- Sybase" or "Database-Oracle" categories. If we examine these classes, we will find that each maintains the interface of its superclasses but implements the specifics for its specific DBMS. For example, the SybaseSession class provides the behavior necessary to maintain a Sybase session. When we initialize a SybaseSession, or ask to prepare an SQL statement, the SybaseSession knows how to allocate the correct structures, make the correct calls, and interpret the results. These classes represent the API wrapper layer.

Finally, we have the ExternalInterface classes called by the wrapper layer. These classes are found in the category "Database-External-Libraries." These classes represent the actual function libraries and provide the means to make the function calls. These methods are called only by the API wrapper layer. Many of the benefits discussed earlier can be seen in this example. There is a common interface to all database systems. The programmer does not need to know the DBMS implementation details to access a database. Furthermore, the database access interface remains the same for all databases, so once the user is familiar with the interface there is nothing to relearn. Exception handling is taken care of by the EXDI classes, with standard errors defined that can be trapped and handled by applications. The database behavior is properly encapsulated. System changes are localized. Database differences are accounted for while nothing is hardcoded. The Smalltalk programmer can be comfortable with the implementation. We also realized an even greater benefit when we needed to access DB2 data. We did not have to write the DB2 access code from scratch because it fit nicely into the provided framework. We were able to achieve DB2 access by creating an ExternalInterface class and then writing the API wrapper Layer classes (DB2Connection , DB2Session, and DB2Transaction) to take care of DB2-specific implementation details. The rest of the implementation we got for free. To top it all off, we did not even need to document the system. All the documentation was already written and published (a programmer's dream!).

SUMMARY

We have shown the approach we take when bringing external resources into the Smalltalk environment. This approach will be useful when you need to bring in any external functionality, especially if you plan on doing so for more than one platform. We have already used it for database access, APPC communication, and to make calls out to the OS/2 environment. There are a few other areas where we see potential for this idea. One could implement a multimedia framework, similar to the EXDI framework. This framework would mask the underlying multimedia implementation and allow subclasses to implement multimedia under both OS/2 and Windows. Other areas of interest might include file dialogs, host menus and widgets, and IBM System Object model (SOM). In our next article we will present a full implementation following the approach we have outlined.

Yoel Newman is a Senior Systems Consultant with American Management Systems (AMS). He can be contacted by email at yoel@aol.com. Michael Parvin is Senior Systems Consultant with Metropolitan Life Insurance. He can be contacted by email at mparvin@tigger.jvnc.net.

Segregating application and domain: Part 1

Tim Howard

THIS ARTICLE IS THE FIRST IN A SERIES of three dedicated to the topic of application and domain segregation in VisualWorks application development. This first article presents the case of why it is essential that an application have a strict segregation between its application information and its domain information. The second article will discuss 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.

We begin with a review of MVC fundamentals, including definitions for application and domain information, followed by a brief history of application development in Smalltalk. Considering this background, it is argued that any VisualWorks application of merit—primarily those with a persistent store—should be designed and implemented with a strict segregation between the application information and the domain information.

MVC FUNDAMENTALS

Before launching into the discussion at hand, it is prerequisite that we back up and cover some MVC fundamentals. The MVC perspective is a way of breaking an application, or even just a piece of an application's interface, into three parts: the model, the view, and the controller. In this context, the term *application* is used to mean one or more windows working in a coordinated and related effort to provide a service to a user community. A word processor is an example of an application—its printer driver is not.

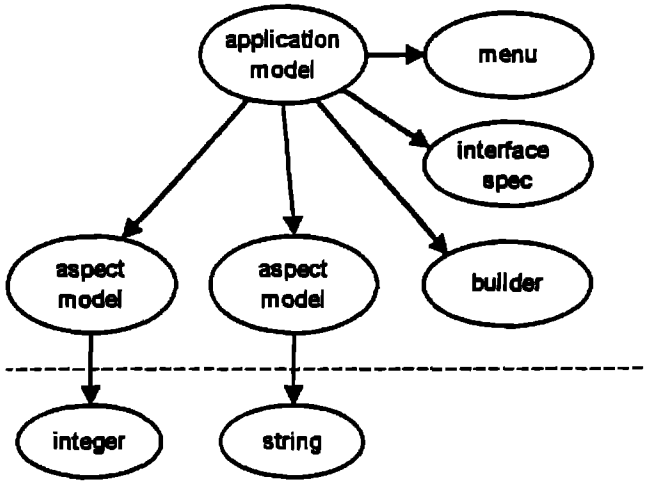


Figure 1. Application model object diagram.

A *model* is an object that manages information. It calculates, sorts, stores, retrieves, simulates, emulates, converts, and does just about anything else you can think of doing to information. As the MVC architecture has matured, it has become apparent that the model's information can be divided into two categories—domain information and application information. A model's *domain information* includes information concerned with the problem space. For example, if we have an airline reservation application, the flight schedules, prices, seating arrangements, and credit card numbers would all be domain information. Each identifiable piece or subset of the model's domain information is called an *aspect*. An aspect can be as simple as a single string or number, or as complex as a subsystem of other interrelated objects. A model's *application information* is any information that is used by the application but is not part of the problem space. In the airline reservation example, error messages, icons, and menus would be part of the application information. A model by itself has no visual representation, nor does it interact with the user or receive any user input.

The *view* provides a visual interpretation of the information contained in the model, which suggests, quite correctly, that there can be more than one view per model. As the information in the model changes, the view should automatically redraw itself to reflect those changes. A view depends on the information contained within its model to fullfill its duties. The *controller* works in conjunction with view and accepts user input—usually keyboard and mouse input. The controller can process this input itself or pass it on to the model or view for processing.

There are certain objects, called *dependents*, that are interested in the information contained within the model and especially interested in changes to that information. A model maintains a collection of its dependents and whenever the model changes *any* aspect of its internal state, it broadcasts a notification of that change to *all* these dependents. It is then up to each dependent to decide for itself if it is interested in the particular change or not. Any object can be a dependent, but the most common dependents are views, windows, and other models (and since the introduction of VisualWorks, DependencyTransformer objects, which do not fit into any of the aforementioned categories).

Any object can be a model because the basic model behavior is implemented in Object. The Model class, however, improves upon this implementation and therefore most models are an instance of some subclass of Model.

Some examples of model classes are ApplicationModel, UIPainter, ValueHolder, PluggableAdaptor, and Browser.

HISTORY OF SMALLTALK APPLICATION ARCHITECTURES

Before VisualWorks, Smalltalk applications employed an architecture often referred to as the *classical MVC architecture.* In this architecture, a single monolithic model (or very few models) assumed most of the model type responsibilities and managed several aspects of information. Such a model had several dependents—a window, other models, and usually a view for each of the aspects. Also, the model's class typically assumed the responsibility of creating the window interface for the model. The System Browser, implemented by the Browser class, is a good example of an application developed with this type of architecture. The main problem with the classical MVC architecture is that a single model manages both application and domain information, and tries to manage the user interface at both the window and component level.

VisualWorks has enhanced the MVC architecture in several ways. Chief among these is the realization that standard model behaviors (i.e. those described in the classes Object and Model) are insufficient for running applications. To manage an entire application, or at least an entire window, a specific type of model is required called an *application model.* The abstract implementation for such a model is described in the ApplicationModel class. An application model manages the application information and leaves the domain information to its aspect models. An *aspect model* manages a single aspect of information and is usually associated with a single interface component. Much of the application information is referred to as interface *resources* and includes such things as menus, icons, labels, and interface specifications. An application model also delegates interface construction to an object called the *builder.* While application model architecture is a great improvement over classical MVC architecture, the domain information contained within the various aspect models is loosely distributed throughout the application, making it difficult to manage. Thus, the domain information is not fully independent of the application model. Figure 1 is an object model of a generic application model.

A model's domain information includes that information concerned with the problem space.

THE NEED FOR DOMAIN SEGREGATION

The complete segregation of domain information from the application information becomes essential when an application of any merit is intended. This is especially true in a client server architecture where the application resides on a client machine but the data, or domain information, resides on a server or is even distributed among several machines. The reasons for such a strict segregation of application and domain information are listed below and subsequently discussed in detail.

- Facilitates persistent storage.
- Provides domain cohesion and logical arrangement of the domain information.
- Keeps application information out of persistent store.
- Keeps dependents out of persistent store.
- Facilitates and abstracts the analysis and design processes.

The application information typically resides on the client machine and has no need to persist outside the virtual image. For all but the most trivial applications, however, the domain information must persist in a database. A Smalltalk virtual image, regardless of the client machine's available resources, soon becomes inadequate as a persistent store for domain information. Furthermore, a virtual image does not offer traditional database facilities such as concurrency control, security, locking, transactions, rollbacks, recovery, and multiuser access. Therefore, the domain information should reside in some kind of persistent store, presumably located on a server machine, where it can be managed by a database management system and be available to several client machines.

The problem with making domain information persist under the current application architecture is that it is scattered throughout the application model. It is contained in various ValueHolders, SelectionInLists, and even within additional, embedded application models in the case of subcanvases or satellite windows. In fact, an application model, by design, is nothing more than a loose confederation of independent models, each operating on its own piece of domain information. As an illustration, suppose we have an application for maintaining employee information for a corporation. Such an application could be filled with all sorts of domain information about the employees—names, social security numbers (SSNs), salaries, supervisors, addresses, dates of birth, etc. Each time we want to store the information for a single employee, we have to traverse all the input fields, lists, text editors, and subcanvases, collect all the relevant domain information for that employee, and ship it all to the database. In the event that we want to fetch the information for a given employee from the database for viewing or editing, we have to access several pieces of information and target each one for its particular aspect model. It would be nice if we had a single handle, or reference point, for all this information. We would like to bundle all the relevant domain information into a single cohesive object—perhaps something like an Employee object that contains all the domain information for a single employee of the company. Ideally, we would like to hand this single domain object to the database for storing, or hand it to an application model for viewing and editing.

One might argue that the application model references all the pertinent domain information, so why not just have it persist in the database. While this is true enough, and

quite appealing at first, upon closer scrutiny we can see that this idea has two serious flaws. First, application information usually does not need to persist in a database. We do not need a copy of the same menu to accompany each Employee object stored in the database. Second, the domain information is very loosely affiliated within the application model. There are too many intermediaries from one piece of domain information to the next.

It is conceivable that certain types of application information might be so large that it is inconvenient, or impossible, to burden each client machine with a copy. A mature help environment is a good example. Such cases require a change in perspective. The help facility becomes an application in and of itself and its domain information is the text and bitmaps comprising the help information. This example is a good illustration of the fact that there is often a gray area between what constitutes domain information and what constitutes application information. It is essential, however, that the design process clearly resolve what is part of the application and what is part of the domain. For example, error codes might be part of the application in one project and part of the domain in another. In either case, however, this must be resolved before implementation. The acid test is "What information must persist?"

Another problem in trying to store the application model in the database is that the application model has dependents, and also references other models—each with its own dependents. We do not want to store anything in the database that has dependents! This is a cardinal rule of persistent objects, the reasons for which are as follows:

- Dependents exist primarily for purposes of the user interface and therefore constitute application information.
- Dependents do not describe domain information. A model does not care how many dependents it has, who its dependents are, or what they are. This relationship has no translation into the domain and therefore does not constitute domain information.
- Dependents exist solely to provide a means of notification during interface operation; they would have no meaning to a persistent object.
- Dependents have a way of compounding the relationships among objects such that a single application model can end up referencing a sizable portion of the virtual image.
- Dependents inevitably reference objects that are known to the virtual machine. Another cardinal rule for persistent storage is to never make an object persistent if it is referenced by the virtual machine. Such references are specific to the client machine, have no meaning in a persistent media, and cannot be accurately reconstituted when the persistent object is fetched from the database.

For the reasons listed above, it is best to leave models and any other objects with dependents out of the database.

Describing the application strictly in terms of the domain information facilitates the analysis and design processes and removes any unnecessary details of interface development. In fact, a good design should exclude application-specific information as much as possible, and concentrate strictly on the domain. Such a design is largely independent of the actual application development and can even be independent of the language of implementation.

SUMMARY

Before we embark on all the work required for adequately segregating the domain information from the application (the following articles in this series), it is important that you understand exactly why we want to do this. The main reasons for segregating the domain information from the application are summarized below.

- Domain information usually resides in a database while application information should stay in the client machine's virtual image.
- Domain information should be bundled into convenient container objects relevant to the problem space.
- Domain information should be clear of any dependent objects and should be completely clear of any ties to the virtual machine.
- Domain information represents the problem space, and design issues should relate as much as possible to the domain information and exclude as much as possible the application information.

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.

Managing concurrency conflicts in multi-user Smalltalk

Jay Almarode

HERE ARE A number of advantages when multiple users access shared objects in a single object space: Users share behavior as well as object state, developers do not have to write mapping code between Smalltalk and a persistent store, and delivering and updating applications is simply a matter of making the changes public. However, when multiple users can view and modify shared objects, there is a potential for conflict.

Concurrency conflicts occur when one user reads an object that another user has modified, or when two users modify the same object. For example, if one user reads an account balance that has been modified and committed by another user, it is imperative that the transaction experience a conflict. This is because any decision made and subsequent code executed is based upon a value that is no longer valid. When the transaction attempts to commit, the attempt is unsuccessful due to the concurrency conflict.

When building applications in single-user Smalltalk systems, developers do not have to consider the possibility of concurrency conflicts on their Smalltalk objects because they can treat all of object memory as their own private domain. Instead, they must map the application's concurrency requirements onto the concurrency control mechanisms provided by some persistent store. In multi-user Smalltalk, the underlying execution engine and transaction manager provide the concurrency control mechanisms. This column will describe the mechanisms for concurrency control in multi-user Smalltalk and describe some techniques for resolving concurrency conflicts.

There are two approaches to concurrency control. One approach is to acquire locks on objects. This approach, called "pessimistic," allows a user to prohibit other users from reading or writing a particular object. Acquiring a lock on an object guarantees that at commit time, certain kinds of conflict will not occur on that object. Locking has its drawbacks, though. When an object is locked, its availability is reduced for other users. Acquiring a lock typically requires the arbitration of a centralized lock manager, which may involve additional network communication in a client/server architecture. And using the pessimistic approach requires that application developers understand which objects will be read or written. For single objects, this might not be too difficult to do. But for networks of objects, it might not be obvious which object will eventually be written when an operation is invoked on the root node in the network.

A second approach to concurrency control, called "optimistic," does not use locking, but instead determines concurrency conflicts when a transaction attempts to commit. With this approach, an application reads and writes objects without explicitly worrying about other users. At commit time, the system determines if any of the objects read or written by this transaction were also read or written by other committed transactions. If so, then a conflict occurs and the attempt to commit fails. This might sound drastic, but in many cases, applications are only reading the majority of objects anyway, so the chance of conflict may not be too high. If an application knows that it will be writing objects also accessed by other applications, it can always acquire locks on the object. The two approaches are not mutually exclusive. If a conflict should occur at the time of commit, the user can find out which objects experienced the conflict and perhaps take steps to resolve the conflict. In SmalltalkDB, the data definition and manipulation language for GemStone, a user can find out which objects experienced conflict by sending the message System transactionConflicts. This message returns a dictionary whose keys are symbols indicating the kind of conflict, and whose values are arrays of objects that experienced the conflict. Using this information, an application can take steps to save information that might be lost when the transaction is aborted (for example, by writing information into a newly created object or by writing data to a file).

In discussing concurrency conflicts on objects, I've discussed conflicts in terms of reading and writing an object at the physical level. Most persistent object-based systems detect conflicts by recognizing when concurrent transactions have read or written the same objects, irrespective of the logical operations that caused those reads or writes. There has been much work in concurrency control for abstract data types that is applicable to object-based systems.[1–3] The main thrust of this work is that even though there may be conflict on the physical level, the logical specification of an object and its operations may allow the physical conflicts to be resolved. For example, two concurrent transactions may add some objects to an instance

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

#'Read-Write'	My transaction read an object that another transaction wrote
#'Write-Read'	My transaction wrote an object that another transaction read
#'Write-Write'	My transaction wrote an object that another transaction wrote
#'Read-ExclusiveLock'	My transaction read an object on which another transaction acquired an exclusive lock
#'Write-ReadLock'	My transaction wrote an object on which another transaction acquired an exclusive lock
#'Write-WriteLock'	My transaction wrote an object on which another transaction acquired a write lock
#'Rc-Write-Write'	My transaction wrote an RC object that another transaction wrote, and the conflict could not be logically resolved.

Table 1. describes the various kinds of conflicts that can occur.

of Bag. The second transaction that attempts to commit will experience conflict since the first transaction wrote the same bag (this is a write–write conflict). However, there is no logical reason why two concurrent transactions cannot add objects to the same bag. If the underlying system can resolve these physical conflicts so that the end result is that the bag contains both transaction's additions, then the second transaction should be allowed to commit successfully. Some systems solve this problem by using locking protocols, but this reduces concurrency by making the bag unavailable for concurrent modifications.

RcQueue	multiple adders to the queue will not conflict a single remover from the queue will not conflict with adders multiple removers from the queue will conflict
RcBag	multiple adders to the bag will not conflict a single remover will not conflict with adders multiple removers of disjoint objects will not conflict multiple removers of the same object will conflict if they attempt to remove more than the number of occurrences in the bag
RcHashDictionary	multiple updaters of entries with disjoint keys will not conflict multiple updaters of an entry with the same key will conflict multiple removers of entries with disjoint keys will not conflict multiple removers of an entry with the same key will conflict readers of an entry will not conflict with updaters of the same entry
RcCounter	multiple incrementers or decrementers will not conflict readers of the counter value will not conflict with modifiers of the value

Table 2. Various RC classes and their semantics.

Listing 1. Example using optimistic concurrency control.

```
"The Bag and The Object are global variables for the
 purpose of this example"

    | addToBag |
    addToBag := true.
    [ addToBag ] whileTrue: [
        " add the object to the bag "
        TheBag add: TheObject.
        " attempt to commit the transaction "
        System commitTransaction
                " if commit was successful, exit the loop "
            ifTrue: [ addToBag := false ]
                " if unsuccessful, abort the transaction and try again "
            ifFalse: [ System abortTransaction ]].
```

Listing 2. Example using locks.

```
    | tryAgain |
    tryAgain := false.
    [
        " Attempt to acquire a write lock on the bag "
        System
            writeLock: TheBag
                " if lock was denied, keep trying "
                ifDenied: [ tryAgain := true ]
                " if lock is dirty, abort the transaction to update your view "
            ifChanged: [ System abortTransaction ].

        tryAgain

    ] untilFalse.

    " at this point, we've acquired the lock on the bag "
    TheBag add: TheObject.
    System commitAndReleaseLocks
```

In using objects in a multi-user setting, a developer must not only think about the functional semantics of an object (what an operation does to an object), but also its concurrency semantics (what concurrent operations are allowed on the object). In SmalltalkDB, the kernel class library has been extended to include classes particularly tailored to multi-user access. These 'reduced-conflict' classes (called RC classes, for short) have functional semantics the same as their single-user counterparts, but have been specifically implemented to provide more concurrency. The cost of this additional concurrency is greater memory usage per object, and potentially slightly longer time to commit. Table 2 lists the reduced-conflict classes and their concurrency semantics.

In SmalltalkDB, programmers have a number of ways they can manage concurrency conflicts. They can lock objects to ensure a successful commit; they can abort their transaction when a conflict occurs and retry operations; they can utilize RC classes when appropriate in their appli-

cations. The following code examples illustrate these three approaches for adding an object to a shared bag. The example in Listing 1 illustrates the optimistic approach. After adding the object to the shared bag, we attempt to commit the transaction. If another transaction has modified the bag, we may get a concurrency conflict, and the attempt to commit will fail. In this case, we abort the transaction, causing the view of the bag to be updated. We can then add the object to the bag and attempt to commit again.

The example in Listing 2 shows one way to acquire a lock on an object. It attempts to acquire a write lock since we know we will be modifying the bag by adding an object to it. It is possible that the lock may be denied because another transaction has already acquired a lock on the bag or because we do not have write authorization for the bag (object authorizations will be the subject of a future column). For this example, we assume we have authorization to modify the bag; otherwise we would add code that checks our authorizations and takes some other course of action. If the lock is denied, we set a boolean flag so that we continue trying to acquire the lock (presumably until another transaction releases its lock). It is possible that the lock may be acquired but a modification to the bag has been committed by another transaction since this transaction began. This is called a "dirty" lock. In this case, we abort the transaction to update our view of the bag, then proceed with our addition to the bag since we continue to hold the lock after the abort operation.

```
" TheBag is now an instance of class RcBag "
TheBag add: TheObject.

"by the concurrency semantics of RcBag, this transaction can
successfully commit "
System commitTransaction
```

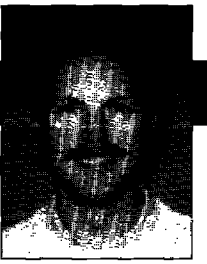
Listing 3. Example using RC Bag.

The final example in Listing 3 illustrates the ease of use of RC classes in SmalltalkDB. Since an adder to the RcBag will not conflict with other adders, removers, or readers of the bag, the transaction will not conflict. The implementation of RcBag uses various strategies to avoid physical conflict on the bag. When a physical conflict does occur, the underlying system attempts to resolve those conflicts if they are determined not to be logical conflicts.

Hopefully this column has given you some insight into managing concurrency in multi-user Smalltalk applications. When multiple users share objects, the application programmer must be aware of the potential for conflict. There are a number of techniques for avoiding concurrency conflicts and when they do occur, the application can take steps to resolve those conflicts.

References

1. Weihl, W. Local atomicity properties: Modular concurrency control for abstract data types, ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS, 11(2), 1989.
2. Herlihy, M. Apologizing versus asking permission: Optimistic concurrency control for abstract data types, ACM TRANSACTIONS ON DATABASE SYSTEMS, 15(1), 1990.
3. Schwarz , P., and A. Spector, Synchronizing shared abstract types, ACM TRANSACTIONS ON COMPUTER SYSTEMS, 2(3), 1984.



Super + 1

Kent Beck

ONCE AGAIN, NO GARBAGE COLLECTORS. I've been busy paying the bills, so I haven't had a change to look in detail at the garbage collectors in the various Smalltalks. I'll get to it, but those college educations have to come first.

Smalltalk Solutions was a blast! Four hundred people packed into the hotel, giving the whole get-together quite a buzz. Of course, that could be because of the hordes of European and Asian tourists. I don't think I got onto an elevator and heard less than three languages the whole four days. Mark this one on your calendar for next year.

I had a great time talking on Wednesday of the conference. My performance-tuning talk was full, with lots of great give-and-take about performance issues. I gave a talk about patterns in the afternoon, and somehow we crammed even more people in. One thing I was uniformly surprised by during my talks was how open everyone was. It's hard to stand up in a room of 250 people and say, "I screwed up thus-and-so; how can I avoid it in the future?" The other thing I appreciated was how much dialog resulted. It wasn't me bringing down the stone tablets, it was more experienced and less experienced people sharing problems and solutions.

The best part of the whole thing was that when I got tired of talking and crowds, I went up to my hotel room and really cranked on code. It's been awhile since I've single-mindedly worked on something just for me. Now I remember why I love programming.

Well, the really best best thing about it was the cheesecake across the street. I must have consumed 10 Kcals having great talks with new friends and old.

And now, some content.

SUPER

A couple of years ago, I published a column about how to use "super" in Smalltalk. It turns out there are only a few legitimate uses, and several common mistakes. I've always liked that column, but I always thought it a shame that I wrote it before I was any good at writing patterns. I'm here to change all that. Because my pattern skills have improved, and because the readership of THE SMALLTALK REPORT has increased so dramatically in the last two years, I'll take another whack at talking about super, this time in terms of patterns. At the end, I'll throw in one more pattern that came up and smacked me in the face recently.

Before I jump in, let me first say that I think inheritance is vastly overrated. It is the least useful feature of the big three (encapsulation, polymorphism, and inheritance). If I had to do without one, inheritance is the one I drop.

However, inheritance is there, and when it is working well it is a joy to use. It results in code that is so highly compressed it is almost like reading poetry. I introduce these three methods and, voilà, I have an object that responds to 30 messages in a new and interesting way. This is the strength and weakness of inheritance. If you don't speak the language of the superclass, there is no way you will understand the subclass.

Three simple rules will keep you out of most of the trouble inheritance can cause:

1. **Keep it in the family**. This is Rick DeNatale's Law of Inheritance. If you are going to subclass, make sure that the superclass is either rock stable or the providers of the superclass are committed to bringing you forward as changes occur. It works best if you own both classes.
2. **Follow the rules**. Factored Superclass tells you to make superclasses only when forced to do so by duplicated concrete implementation, not merely speculation about the nature of the universe. The "super" patterns that follow help reduce coupling. Composed Method, when used in the superclass, ensures that subclasses needn't duplicate code.
3. **Never refactor a hierarchy twice in a row**. Early in my career I wasted more time twisting inheritance hierarchies this way and that, trying to share one or two more lines of code. If you do refactor an inheritance hierarchy, live with it for a while the new way. Be prepared to dump the refactoring if it doesn't go well.

PATTERN: SUPER

How can you invoke superclass behavior?

An object executes in a rich context of state and behavior, created by composing together the contexts of its class and all its class' superclasses. Most of the time, code in the class can be written as if the entire universe of methods it has available is flat. That is, take the union of all the methods up

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).

the superclass chain and that's what you have to work with.

Working this way has many advantages. It minimizes any given method's reliance on inheritance structure. If a method invokes another method on self, as long as that method is implemented somewhere in the chain, the invoking method is happy. This gives you great freedom to refactor code without having to make massive changes to methods that assume the location of some method.

There are important exceptions to this model. In particular, inheritance makes it possible to override a method in a superclass. What if the subclass method wants some aspect of the superclass method? Good style boils down to one rule: say things once and only once. If the subclass method were to contain a copy of the code from the superclass method, the result would no longer be easy to maintain. We would have to remember to update both (or potentially) many copies at once. How can we resolve the tension between the need to override, the need to retain the illusion of a flat space of methods, and the need to factor code completely?

> Invoke code in a superclass explicitly by sending a message to "super" instead of "self." The method corresponding to the message will be found in the superclass of the class implementing the sending method.

Always check code using "super" carefully. Change "super" to "self" if doing so does not change how the code executes. One of the most annoying bugs I've every tried to track down involved a use of super that didn't do anything at the time I wrote it, and that invoked a different selector than the one for the currently executing method. I later overrode that method in the subclass and spent half a day trying to figure out why it wasn't being invoked. My brain had overlooked the fact that the receiver was "super" instead of "self," and I proceeded on that assumption for several frustrating hours.

Extending super adds behavior to the superclass. Modifying super changes the superclass' behavior.

PATTERN: EXTENDING SUPER

You need to extend superclass behavior.

How do you add to a superclass' implementation of a method?

Any use of super reduces the flexibility of the resulting code. You now have a method that assumes not just that there is an implementation of a particular method somewhere, but that the implementation has to exist somewhere in the superclass chain above the class that contains the method. This assumption is seldom a big problem, but you should be aware of the trade-off you are making.

If you are avoiding duplication of code by using super, the trade-off is quite reasonable. For instance, if a superclass has a method that initializes some instance variables, and your class wants to initialize the variables it has introduced, super is the right solution. Rather than have code like:

```
Class: Super
Superclass: Object
Variables: a
Super class>>new
    ^self basicNew initialize
Super>>initialize
    a := self defaultA
Class: Sub
Superclass: Super
Variables: b
Sub class>>new
    ^self basicNew
        initialize;
        initializeB
Sub>>initializeB
    b := self defaultB
```

where the subclass has to invoke both initializations explicitly, using super you can implement:

```
Sub>>initialize
    super initialize.
    b := self defaultB
```

and not have Sub override "new" at all. The result is a more direct expression of the intent of the code—make sure Supers are initialized when they are created, and extend the meaning of initialization in Sub.

> When you want to extend the meaning of a superclass method, override the method and invoke "super" as either the first or last statement of the method.

PATTERN: MODIFYING SUPER

You need to modify a superclass' behavior.

How do you change the part of the behavior of a superclass' method without modifying it?

This problem introduces a tighter coupling between subclass and superclass than Extending Super. Not only are we assuming that a superclass implements the method we are modifying, we are assuming that the superclass is doing something we need to change.

Often, situations like this can best be addressed by refactoring methods with Composed Method so you can use pure overriding. For example, the following initialization code could be modified by using super.

```
Class: IntegerAdder
Superclass: Object
Variables: sum, count
IntegerAdder>>initialize
    sum := 0.
    count := 0
Class: FloatAdder
Superclass: IntegerAdder
Variables:
FloatAdder>>initialize
    super initialize.
    sum := 0.0
```

A better solution is to recognize that IntegerAdder>>initialize is actually doing four things: representing and assigning the default values for each of two variables. Refactoring with Composed Method yields:

```
IntegerAdder>>initialize
    sum := self defaultSum.
    count := self defaultCount
IntegerAdder>>defaultSum
    ^0
IntegerAdder>>defaultCount
    ^0
FloatAdder>>defaultSum
    ^0.0
```

However, sometimes you have to work with superclasses that are not completely factored. You are faced with the choice of either copying code, or using super and accepting the costs of tighter subclass/superclass coupling. Most of the time the additional coupling will not prove to be a problem. Communicate your desired changes with the owner of the superclass. In the meantime:

> When you want to modify the meaning of a superclass method, override the method and invoke "super" as either the first or last statement of the method.

COMMENTS

Here is where an interesting point about patterns comes in. Notice that these two patterns only tell you to invoke "super" with the same selector as the currently executing method. The original article discussed a couple of cases where it was marginally useful to invoke super with something other than the currently executing message selector. In trying to translate them to patterns, I wasn't convinced that they were actually good style, and they were terribly rare. Rather than write poor patterns that wouldn't be used often, I chose to leave them out (go browse all users of super in any stock image if you want to find how super is misused).

PLUS ONE

Here is a pattern morsel I'll throw in, mostly because I was so embarrassed recently when I missed it, and it took my clients to point out how much easier life would be once I reintroduced it.

Let me set the stage. I am writing a framework for this client that invokes one of many subclasses that they are writing. The protocol has been pretty unstable for a while, with names changing and parameter lists changing as we matured the framework. This resulted in the need for more communication than is productive, and slowed their development.

Now me, I'm willing to go through lots of pain to get to the right solution. If I have to go change 25 selectors because I found a better word for something, I'll do it. My

assumption is always that the improved communication and resulting reduction in lifecycle cost is always worth the effort. In this case, my "self sacrifice" got the best of me. If I'd used Parameters Object about two months ago, the whole project would have sped up by about a week. Sigh...If only my computer would quit reminding me how little I really know.

PATTERN: PARAMETERS OBJECT

How can you best write methods with many parameters?

Reducing the coupling between objects is good. Eliminating direct references from one object to another lets you use the two objects more independently. You can replace most direct references by passing extra parameters.

Going too far down this road leads you to code that doesn't communicate well. There are times when the communication between two objects is so pervasive, such an important part of your conception of the program as a whole, that you can't imagine not having a reference one to another. A Rectangle needs its Points. Further, even where you might be able to replace a direct reference with a parameter, passing extra parameters leads to difficult formatting and naming decisions and obscures the intent of the methods behind the host of keywords required.

If you have decided that you don't want a direct reference, but you still need several parameters, what do you do? The problem becomes worse if there are many implementations of the selector. During development, as you discover the need for more or fewer parameters in certain cases, you have to go around adding and deleting keywords from selectors in many classes.

In a collaborative environment, this redesign is unlikely to ever take place. One strategy is to pass every possible parameter everywhere on the off chance that it might be useful some day. This results in many messages being more complex than they need to be, obscuring the true intent of the code for later refinement or communication to others. The other strategy is to use global variables to short-circuit disciplined communication, thereby reducing the possibility that the code will ever be valuable on its own.

We need a way to decouple instability in the parameter list from instability in the protocol. As protocols change, they should change because of changes in intent. As the list of parameters change, the protocols shouldn't change just to accomodate the need of some particular implementation for extra information.

If you have three or more parameters that are passed three or more levels, or that are passed to five or more implementations of the same selector, create an object with one variable per parameter. Create an instance of the object in the highest-level sender and pass it around.

You may be able to use Composed Method to move computations into the Parameters Object. Do so without regard to whether it "makes sense." If you send two or more messages to the Parameters Object in a single method and then compute with the results, move the computation.

For example, suppose we didn't have Rectangles. Everywhere we compute with Rectangles we have to pass four parameters:

```
...boundsTop: topInteger left: leftInteger bottom:
bottomInteger right: rightInteger...
    ...area := bottom - top * (right - left)...
```

Introducing Rectangle as a Parameters Object, we now have:

```
...bounds: aRectangle
    ...area := aRectangle bottom - aRectangle top *
(aRectangle right - aRectangle left)...
```

Far better to move the computation close to the data:

```
Rectangle>>area
    ^bottom - top * (right - left)
...bounds: aRectangle...
    ...area := aRectangle area
```

The resulting code is much more flexible, because we can change the implementation of area computation to suit the needs of the client without having to touch the client's code.

Another common implication of this pattern is that the method may be relying on sending messages directly to the parameters before you introduce the Parameters Object. Use Simple Delegation in the Parameters Object to hide its existence from the method.

Between these two techniques, you will often find that the Parameters Object takes on an important role in the whole computation. These are the kinds of objects that thoughtful analysis will never reveal. As valuable as they are, you will only find them if you listen to what your program tells you.

Mark Lorenz

Model integrity through custom instantiation

CREATING INTELLIGENT OBJECT MODELS of your business that serve your software needs is what object technology is all about. We would like these models to be as robust as possible. One technique to ensure the integrity of your object model is by instantiating your objects with their essential relationships already established. You do this by defining custom instantiation class methods—a technique I call *instantiation integrity*.

A MODEL...

Let's say we decide Figure 1 represents a portion of our business' objects and their important relationships. A SalesTransaction has one Person associated with it and contains one or more LineItems. A LineItem is associated with one kind of Product.

In particular, let's assume that our business rules require that:

- a SalesTransaction cannot exist without a related customer (*a* Person), because a customer of the Store must purchase something to have a SalesTransaction. We'll ignore complications such as other businesses buying from our Store for now.

SalesLineItems cannot exist without a related Product, because their reason for being is to document the quantity and price of a certain Product being sold within a particular SalesTransaction. We will not discuss the many implications and complexities of something seemingly as simple as figuring out a price, such as price groups and sales events.

Listing 1 shows some possible client code that creates LineItems and adds them to a SalesTransaction.

What happens if one of our clients forgets to set newLineItem's product? Our model would be in an invalid state, according to our business rules. All clients are also required to write more code to create a SalesLineItem. Let's see how we can ensure these relationships exist upon creation of our object instances.

...WITH INTEGRITY

As you know, the new class method exists to instantiate objects in our image. We can use this method to get an "empty" instance—one that has all state initialized to nil. What we want to create are instances that have valid state immediately set for all clients of a class. We do this by defining custom class instantiation methods.

Listing 2 shows an example method to create SalesTransactions that have their related customer set immediately upon instantiation.

Client code might look like:

```
newOrder := SalesTransaction for: myCustomer.
```

Notice that the instance of SalesTransaction is in a valid business state immediately, in this case by having a related customer. It is likely that we will have fewer problems in developing and maintaining a system built using this design strategy. Clients also have less work to do.

Similarly, Listing 3 shows an example of how we can

Listing 1. Sample client code without instantiation integrity

```
...
newLineItem := LineItem new.
newLineItem product: self selectedProduct.
mySalesTransaction addLineItem: newLineItem.
...
```

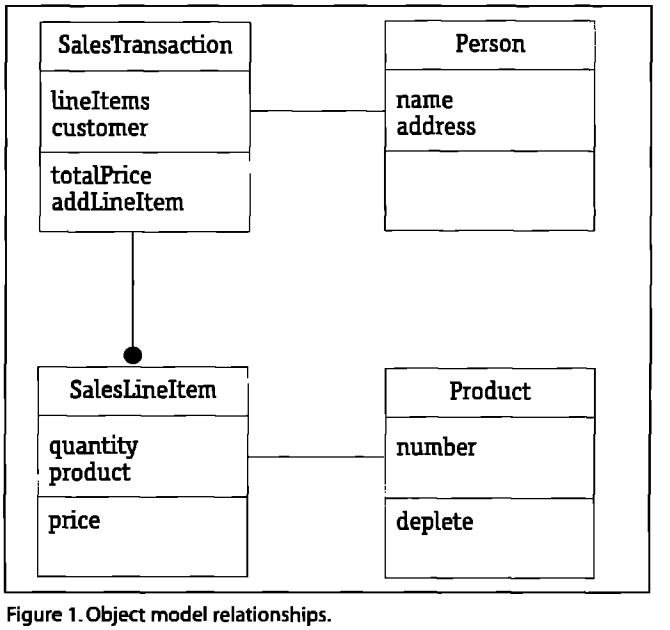


Figure 1. Object model relationships.

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.

Listing 2. Custom instantiation class method for the customer relationship.

```
SalesTransaction class
for: aPerson
        "return an instance of myself with my customer
        set to aPerson"

    ^self new
        customer: aPerson;
        yourself
```

Listing 3. Custom instantiation class method for Product relationship.

```
OrderLineItem class
for: aProduct
        "return an instance of myself with my product set
to aProduct"

    ^self new
        product: aProduct;
        yourself
```

create SalesLineItems with a class method that establishes the relationship to a Product.

The name of the instantiation method depends on the relationship(s) being established. For example, an Amount instance of a Currency might be initialized by:

```
amount := Amount value: aFloat of: aCurrency.
```

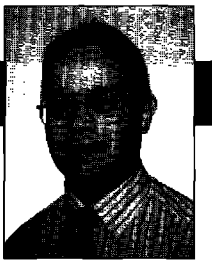
Be careful to include yourself at the end of your cascaded initialization messages. This will ensure that you return an instance of the proper type of object, instead of the last object returned from your initialization methods. In Listing 3, if yourself had been left off, a Product instance would be returned instead of a SalesLineItem instance.

SUMMARY

We have discussed a useful technique for helping to ensure the integrity of our object models. The technique uses custom instantiation class methods to create our objects with their essential relationships immediately established. While this technique does not prevent bad object model states, such as are caused by passing bad parameters, it goes a long way toward placing the business knowledge where it belongs—with the key model classes. This helps each and every client use the model services more safely and effectively.

Terminology

- **Class method:** a method defined for and used by a class instead of an instance.
- **Instantiation**: the allocation of memory for the unique and private state of an instance of a class.
- **Object model**: classes and their relationships as defined by the business requirements.



Math, Part 1

Alan Knight

THERE ARE A NUMBER OF MATHEMATICAL ISSUES that come up very frequently on comp.lang.smalltalk. Most of these issues are language-independent, but because so many people are asking these questions in a Smalltalk context, I think it's important to address them. I'm not a mathematician, so I'm going to try and stay with the simple stuff, and with things that are Smalltalk-specific whenever possible.

I've made this a two-part column. The first explains some of the problems and the second attempts some solutions.

THERE'S A SERIOUS ARITHMETIC BUG...

It seems that every few months there's a post like the following, from Xavier Alvarez (alvarez_x@jpmorgan.com):

> **In VisualWorks, if you enter the following in a workspace**
>
> **100.9 - 100.0**
>
> **and evaluate it, you will get a fabulous result**
>
> **0.900002**
>
> **If you have a solution please tell us! We are building a critical application and need to subtract correctly. I think this is a very serious flaw in PP Smalltalk. Is there a patch for this?**

Although Smalltalk doesn't agree with basic arithmetic in this case, it isn't a flaw in ParcPlace's products, it isn't a problem with Smalltalk in general, and it's not a bug. It's a feature. This is the way floating point calculations work on digital computers. The misunderstanding arises because floating point numbers use a limited number of bits to implement the infinite-precision abstractions we learned in school. Most of the time they look the same, but sometimes the limitations show through.

Representing numbers

Integers work. We can exactly represent any integer in the computer and we can manipulate them without introducing any errors. Some of them may take a bit of space and be a little slow to manipulate, but there are no absolute restrictions. Unlike many languages, Smalltalk doesn't impose a maximum size on integers. We can write expressions like

```
60 factorial / ((60 - x) factorial * x factorial))
```

and get the right answer even though the intermediate results don't fit in 32 bits, or even 32 bytes.

Fractions work, because we can represent them with an integer for the numerator and another for the denominator.

With integers and fractions, we have most of the numbers we need. There are still lots of numbers (infinitely lots) that we can't represent, but these are numbers like π, e, and the square root of 2. These generally don't arise if we stick to simple arithmetic, so we'll ignore them for now. We have bigger problems.

Although Smalltalk allows arbitrarily large integers, it has to run on digital computers, which are optimized for dealing with small numbers. Operations on very large numbers are much slower.

How small? About 4 billion on most machines, and nowadays many can comfortably handle 18 sextillion. That may not seem small, but it's very easy to exceed that limit, particularly if you're using fractions. It doesn't take many operations before the numerators and denominators get very large, and the operations are particularly slow because the fractions need to be reduced.

How slow? I did a couple of very crude benchmarks in Digitalk's V/Win32, and for a simple addition test I found LargeIntegers to be about 100 times slower than SmallIntegers and Fractions with LargeInteger components to be about 25,000 times slower than SmallIntegers. These aren't very reliable benchmarks, and I wouldn't put any faith in the details of the results, but they do indicate that the slowdown is very significant.

Floating point

There's an alternative, which is to use floating point numbers. These use a fixed number of bits, but have an additional exponent which gives the scale of the number. These mirror the scientific notation for numbers, e.g. $1.356 * 10^{12}$. With floating we can represent both 1.5 and 1.5 sextillion with perfect accuracy. We can also manipulate these numbers very quickly, particularly if we have a floating point co-processor.

The drawback is that we've given up absolute accuracy. We have a fixed number of digits, and although we can represent 1.5 sextillion perfectly, we can't handle 1.5 sextillion and 1. There's an even worse limitation on accuracy to do with repeating decimals. These are fractions that can't be represented in any finite number of digits. The most common example is one-third, whose decimal representation is 0.333333... with an infinite number of 3's. If we actually write the decimal form or represent it in a

Alan Knight battles the forces of numerical instability with 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.

computer we will have to truncate it to a finite number of digits, introducing inaccuracy.

One-third is a repeating decimal in base 10, but computers operate in base 2, which has some other problem numbers. In particular, one-tenth cannot be represented with a finite number of binary digits. This is a problem if we want to do really accurate base 10 arithmetic because 0.1 or 100.9 can only be stored as approximations.

For a lot of calculations these inaccuracies are quite acceptable. We're often dealing with input data that is uncertain, and a small loss of precision in exchange for an enormous speed-up can be a good trade-off. This is particularly true in scientific and engineering computations, where floating-point numbers are widely used. It's probably not true in applications dealing with money, where accuracy is extremely important. A good rule of thumb is:

Don't use floats to represent money.

Using double-precision to represent money just means the errors won't show up until the amounts of money are very large.

Hidden errors

Although most people accept the idea of using approximations, they don't like to be reminded of it. If a calculation should yield an exact or easily checkable result, people react very badly to small errors. They may not notice if the cosine of 45 degrees comes out as 0.7072103 instead of 0.70710678, but they're not happy if the cosine of 90 degrees comes out 0.99986452 instead of 1.

For this reason, people designing mathematical software work very hard to make the errors invisible. One way to do this is to adjust the approximations so that the exact answers come up in places that people are likely to notice. Pocket calculators use approximations for functions like sine and cosine, but they are carefully tuned to give the exact answers at places where they're well-known, even at the cost of slightly greater inaccuracy in other places.

The other thing systems do to look more accurate than they are is to round numbers so that errors in the last few digits aren't visible when the number is displayed. All systems do this to some extent, which is why numbers like 0.1 print properly, even though they cannot be exactly represented in the computer. In fact, with double precision floats, most Smalltalks print a maximum of 8 digits when there are 14 or more available, potentially masking quite a lot of round-off error. If your calculations are numerically stable, the errors may never show up in the

printed representation at all. Unfortunately they're still there, and can show up in more subtle ways.

Equality tests

One place that problems arise is comparing floats for equality. Expressions that you'd expect to be equal probably aren't. For example:

```
0.1 + 0.1 = 0.2  ==> true
0.1 + 0.1 + 0.1 = 0.3  ==> false
```

The reason is that floating point equality tests that both numbers are exactly equal. If even a single bit is different the comparison will return false. In the case above, adding 2 floats together doesn't introduce enough error, but adding 3 floats together does. Another good rule of thumb is:

> Don't compare floats for equality.

What you usually want to know is if two floats are close enough together that they can be considered equal. We can find this out by testing a range. For example, we might define a float operation:

```
closeEnoughTo: aFloat
    ^(self - aFloat) abs < 0.00001
```

This works well as long as the precision of the numbers is much larger than the range we're testing against. It doesn't solve all the problems, however. This is not an equality operation in the traditional mathematical sense, and standard assumptions may not hold. For one thing

```
(a closeEnoughTo: b) and: [b closeEnoughTo: c]
```

does not imply

```
(a closeEnoughTo: c)
```

Subtraction

Equality testing exposes minor numerical errors in calculations. Far worse than this is subtraction, which can make calculations very numerically unstable. In particular, subtraction of nearly equal quantities is bad. In numerical analysis, a lot of effort is devoted to re-arranging equations to avoid these kind of subtractions.

The problem is that these subtractions throw away lots of significant digits, magnifying the existing errors. The error in 1.000032 is inconsequential, but if we subtract 1.0 we have nothing left but error. This is what happened in the original post, where subtracting 100.0 from 100.9 multiplied the existing approximation error by 100, making it visible in the final result.

I never noticed this in...

Even though these principles apply to all languages, they may be more or less noticeable depending on characteristics of the platform and the environment. One characteristic is the size of floats. Floats come in two main sizes, single-precision (32 bits) and double-precision (64 bits), with Macintosh also supporting extended precision (80 bits). C, the language most commonly used for implementing Smalltalk, uses mainly doubles, with some half-hearted support for single-precision. Perhaps inspired by this example, most Smalltalks only support one precision, the highest available on the machine. As far as I know, ParcPlace is the only Smalltalk implementation to support both single and double-precision floats, with single-precision the default.

Smalltalk doesn't agree with basic arithmetic in this case, but it's not a bug, it's a feature.

If there's a bug here, it's a public-relations bug on the part of ParcPlace. By making single-precision the default, they've made these errors much more evident than in other dialects. In fact, representatives of two different Smalltalk vendors posted articles saying that their implementation got the right answer, the implication being that it was indeed a bug in VisualWorks. This is, of course impossible. What actually happened is that these implementations used large enough floats that the printed representation of the answer looked right. That is

```
100.9 - 100.0 ==>  0.9, but
100.9 - 100.0 = 0.9 ==> false
```

More precise floating point representations can certainly help the accuracy of calculations, but they can also lead to dangerous complacency, because more sophisticated testing may be needed to catch numerical problems. Using double-precision to represent money just means the errors won't show up until the amounts of money are very large. Susan Stepney (susan@logcam.co.uk) writes:

> **... I'd like to pick up on the word "critical" in the original post. I was taught by an old hand at numerical analysis that "double precision is a crutch used by people who don't do their numerical analysis properly"**
>
> **If your application is *critical*, make sure your numerical algorithms are so *robust* that it doesn't *matter* that you've got 6 figure accuracy.**

What to do about it

It's all very well to talk about the problems with floating point arithmetic, but what we really need are solutions. How do we represent $12.53 if we can't use floats for money? Fortunately, there are a variety of ways to work around these problems, and I'll talk about them in the next issue.

HP Distributed Smalltalk:
CORBA-compliant distributed objects

Jim Haungs

HEWLETT-PACKARD'S DISTRIBUTED SMALLTALK (DST) is a set of extensions to ParcPlace's VisualWorks that enable cooperative processing among objects distributed over a network. Release 4.0 makes DST compatible with VisualWorks Release 2.0. It is fully compliant with the Common Object Request Broker Architecture (CORBA) Release 1.1 adopted by the Object Management Group (OMG), and it supports release 1.0 of OMG's Common Object Services (COS).

DST is composed of several distinct functions organized into three somewhat indistinct layers: the remote procedure call (RPC) transport layer; the CORBA compliance layer, consisting of the Interface Definition Language (IDL), Object Request Broker (ORB), and Common Object Services (COS); and, finally, the HP Desktop layer. I'll discuss each of these in turn.

RPC LAYER

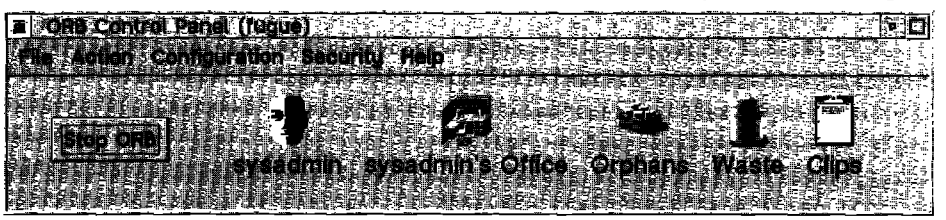
The most basic layer provides remote procedure calls (RPCs) between Smalltalk images. Any Visual Works platform that supports sockets and a TCP/IP stack can be used with DST. However, the current release requires its host to have a fixed IP address, which makes it difficult to use on machines with SLIP or PPP connections; this limitation may be remedied in a future version.

The images may be on the same machine, or they may be distributed over multiple machines in a network. A remote procedure call, or in Smalltalk parlance *remote message passing*, is somewhat more complicated than a normal method invocation because the parameter values must be passed between address spaces. It is not sufficient to push the values on the stack, assuming that both sides of the call can access the same memory. Instead, the parameters must be *marshalled* for transmission, i.e., the objects are traversed and their contents converted into a stream of bytes suitable for transmission over a network. On the other side of the call, the parameter values must be unmarshalled and the objects they represent must be reconstituted. (This is analogous to the processing that takes place for BOSS files: the objects are transitively unraveled, converted to a bytestream, and written to the file; when the contents are retrieved, the opposite occurs.) On the receiving side of the call, the RPC layer must reconstitute the arguments, then effect a local dispatch to the target method, passing it the reconstituted parameter values. When the method has completed, the return value must then be marshalled and sent to the calling machine, where the return value is unmarshalled, reconstituted, and returned to the caller of the remote method.

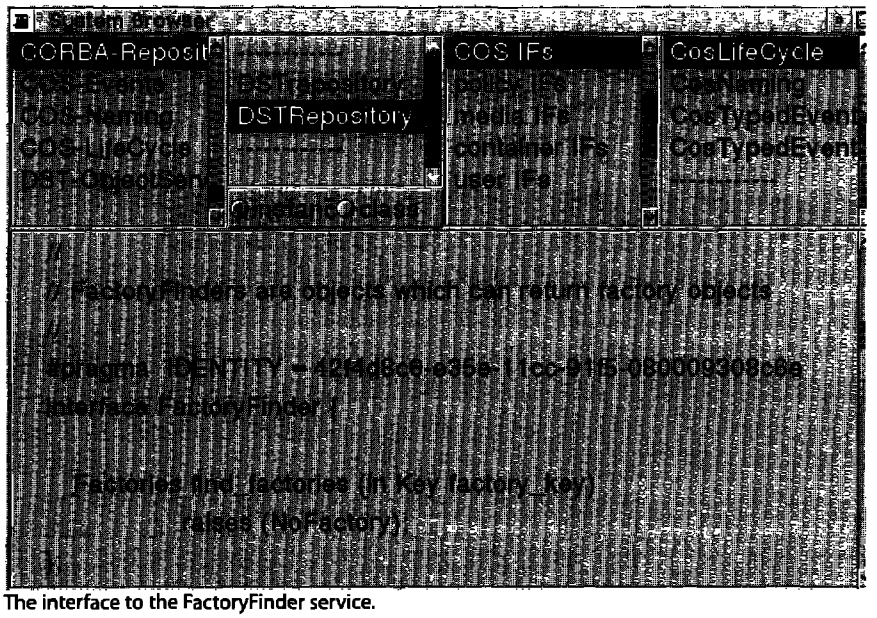
THE OBJECT REQUEST BROKER LAYER

The next layer of DST is the CORBA implementation. The CORBA specification is quite intricate because it addresses a number of complex issues without restricting the style and form of a compliant implementation. Release 4.0 of DST is fully compliant with Release 1.1 of the CORBA specification, and Release 1.0 of the Common Object Services. I'll briefly discuss each of these standards and describe their implementation in DST.

At its most basic level, the ORB architecture is designed for interoperability between languages. To this end, OMG specifies an IDL that describes abstract interfaces for sets of procedure calls. The syntax of IDL is patterned after the declaration syntax of C++, with some additional information provided for distribution, exceptions, and inheritance from other interfaces. At first glance, one might question the value of statically typed interfaces for Smalltalk, but the purpose of the IDL is to describe interfaces in a language-independent way. There is no more need for IDL between two Smalltalk clients than there is between two C clients—they each speak their native language, and barely understand the other. However, sharing an IDL interface enables a CORBA-compliant C program to communicate with a CORBA-compliant Smalltalk pro-



The ORB Control Panel.



The interface to the FactoryFinder service.

gram. In actual use, DST requires you to describe an IDL procedure interface for every method call that can be executed remotely; in practice, this represents a high degree of programming overhead, and is overkill for Smalltalk-to-Smalltalk communication. But as more CORBA implementations are developed, it will become increasingly important to support interlanguage communication. As the tools improve and the need increases, this overhead will decrease with time.

CORBA Interfaces are contained in modules; a CORBA module can contain any number of interfaces. Within an interface, you can define any number of procedure calls. An interface corresponds roughly to a protocol in Smalltalk; it represents a group of related procedure calls that can or should be grouped logically as a unit. Interfaces can be shared among classes, provided the selector names are the same and the parameter types match.

DST has a very nice implementation of the CORBA repository idea, where interfaces can be browsed and edited with the normal Smalltalk source code browser. Even though the IDL syntax is different from Smalltalk, normal editing and compiling is supported by VisualWorks' flexible compilation framework. The DSTrepository class contains one module per method; accepting the source code in a browser invokes the IDL compiler instead of the Smalltalk compiler. Errors are highlighted one at a time, just as in Smalltalk, and ultimately, when the compilation is successful, the IDL module is compiled into Smalltalk.

There are a few things about building cross-platform interfaces that are not so simple. DST relies heavily on the notion of a universally unique id (UUID) that is an encoding of the host IP address and the current date and time. Every interface has a UUID, and inter-machine communication depends on these IDs matching. If you want to invoke a remote procedure in an interface, you must have syntactically correct and semantically matching interfaces on both machines, and their IDs must match. Unfortunately, the implementation of UUIDs leaves much to be desired. Their format is a 34-character string of hexadecimal numbers. To create a new one while editing an interface module, you must type the text "ORBObject newId," highlight the text, select Print It, and then paste the resulting text into your interface definition. One of the most common errors is getting the UUID wrong. I would like to see this process automated and made invisible to the programmer; it is far too error-prone.

The design of a class and the design of its interfaces go hand in hand. Once the class is designed and the methods are coded, you can execute the text "yourClassName asIDLDefinition," which will create a skeletal interface module with an IDL procedure definition for each method on the class. Once the methods are coded, you must create a couple of methods for your class that identify it to the ORB. The CORBAName method returns a Symbol that identifies the module/interface pair that interfaces with the class. The abstractClassId method returns a UUID that uniquely identifies the class. Abstract class IDs are used to create remote instances of a class. All interface UUIDs and class UUIDs must be different from each other, and the corresponding interface IDs and class IDs must match on all the machines that will be communicating.

The next layer up from the basic RPC mechanism is the Object Request Broker. The ORB is an active entity that manages the RPC flow into and out of an image. It runs as a background process in every image that communicates with other images. It is the ORB that first receives a communication from another image, looks up the interface ID, and routes the message to the appropriate object.

In addition to routing messages, the ORB also provides the CORBA Naming Services (NS). Naming is a directory service used to locate remote objects. Names are complex entities composed of a sequence of components and a fixed name. The sequence can be arbitrarily long, and can represent naming schemes from the application domain, machine names, directory paths, database traversal paths, or any arbitrary means of locating an object. Application developers can easily design their own naming services to locate items of interest on remote machines.

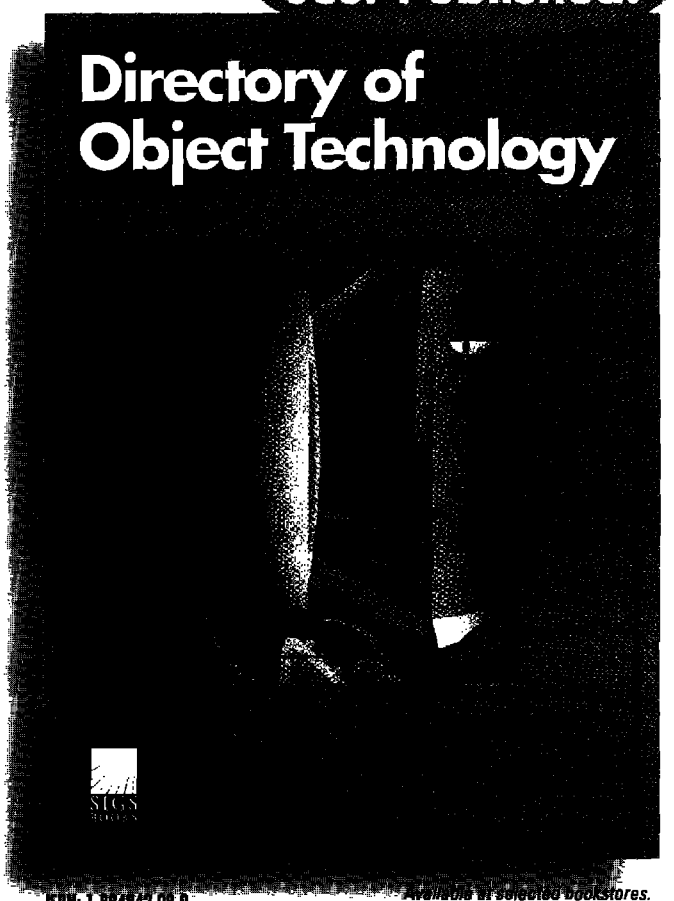
An important distinction in the CORBA spec is between basic services and implementation enhancements. One of the DST enhancements is a clean separation between the semantics of an application and its presentation. This split is similar to the MVC paradigm in Smalltalk, in that it provides for multiple presentations on

Now you can have the entire OT marketplace at your fingertips...

Just Published!

Directory of Object Technology

- ▶ Up-to-date and complete
- ▶ Detailed yet easy-to-read
- ▶ International in scope



ISBN: 1-884842-08-9

Available at selected bookstores.
Distributed by Prentice Hall.

Divided Into 5 User-Friendly Sections — 2,500 Total Entries

1. **Products** — 1309 alphabetized listings by product category. Each entry is carefully described including language and platform.
2. **Services** — 731 alphabetized entries by category. Includes organization name and service offerings.
3. **Training & Mentoring** — 200 alphabetized contact entries by primary focus and specialization.
4. **Books** — nearly 500 titles published during the past decade.
5. **Company Listings** — contact, description, pricing, and platforms of 330 OT-related companies worldwide.

The **Directory of Object Technology** is the one complete resource guide available. *Priced at just $69*, this much-needed sourcebook presents detailed information on every O-O related company, product, and service currently on the market. It contains everything you need to contrast and compare products — helping you make well-informed purchasing decisions.

ORDER TODAY!

❑ Yes! Please rush me the **Directory of Object Technology '95** (ISBN: 1–884842–08–9) at the following rate:

_____ Individual Rate: Just $69 each

_____ Corporate Library Rate: $169 each

Method of Payment

❑ Check enclosed (Payable to SIGS Books)

❑ Bill me/My company

❑ Charge my: ❑ Visa ❑ MasterCard ❑ Amex

Card#:_____ Exp. date:_____

Signature:_____

Postage and handling: U.S. orders add $5 for shipping/handling; Canada and Mexico add $10; Outside North America add $20.

Note: New York State residents must add applicable sales tax. Please allow 2–3 weeks for delivery.

RISK FREE OFFER

If you are not completely satisfied with this product, you may return it within 14 days and receive a complete refund.

SHIP TO

Name_____

Title_____

Company_____

Address_____

City/State/Zip_____

Country/Postal Code_____

Phone_____ Fax_____

TO ORDER:

Return this coupon by

FAX: 609.488.6188

MAIL: SIGS Books P.O. Box 99425, Collingswood, NJ 08108-9970

or order by PHONE: 609.488.9602

SIGS BOOKS

195FPDRA

a common model. HP has taken the MVC paradigm a bit further by allowing presentations and semantics to exist on separate machines. Because the coordination this entails is considerably beyond the MVC change mechanism, HP introduces two new class frameworks for distributed applications: the DSTapplicationObject class supports the application domain model, and the DSTpresenter class supports the presentation layer. By convention, domain models inherit from DSTapplicationObject, and their class names end with the letters SO (for semantic object); similarly, presentations inherit from DSTpresenter, and their names end with the letters PO (for presentation object). The HP Shape example uses two classes: ShapeSO and ShapePO. It is possible to implement remote updates using the standard Smalltalk MVC messages, but it is considerably less efficient than the DST framework, because MVC messages take two roundtrips, one for the #changed message and one for the #update messages. The DST framework accomplishes this more efficiently, but at a cost of having to learn yet another MVC-like framework.

OBJECT SERVICES LAYER

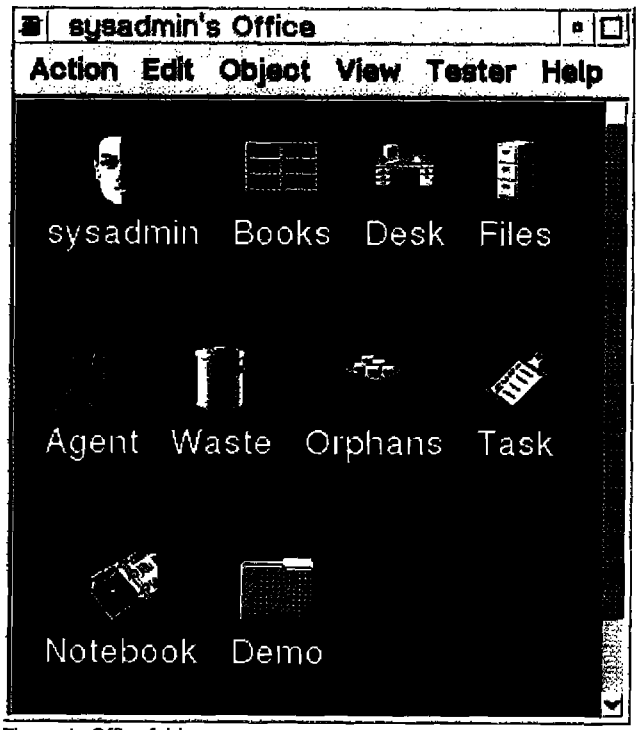
The next layer up from the ORB is the Common Object Services layer. The standards for these services are not as rigorously defined as the IDL interface. HP calls these services the *Object Lifecycle services;* they provide a framework for relating objects via links. Links are used to create *compound objects*, i.e., graphs or networks of related objects that can be manipulated as a unit. Compound objects can be copied and moved between machines, and they can be destroyed as a unit. There are four types of links, in order of descending strength: *containment* links, used to represent concepts like files contained in a directory; *reference* links, which guarantee the existence of linked objects; *designation* links, which don't guarantee the existence of linked objects; and *weak* links, in which an object points to a target object but the target is unaware of the link. The strength of the link is inversely related to fault-tolerance and flexibility. Containment links are the strongest, and they force all the objects in a containment relationship to be collocated, i.e., on the same machine; containment links are strictly hierarchical. The other link forms can represent inter-machine references whose existence is not guaranteed, and which rely on the stability of the underlying network. All objects in a containment relationship can be expected to be accessible if any of them are. The same cannot be said of the other types of relationships. Depending on the nature of the application, the link hierarchy provides for nearly any combination of strength and flexibility the application requires.

The OMG recently approved a new service specification called *Relationship Services*, which completely subsume the functionality currently provided by links, but in a much more general way. Unfortunately, it is more of an entity-relationship model than an object model; but nevertheless, it is much better than the current Link Services specification.

Another CORBA service is the Event Service. Events represent a simpler information flow than procedure calls, but can be configured in more complex ways. Events are triggered by an event supplier, and are received by an event consumer. Orthogonally, consumers and suppliers can either push or pull events. A push event is unsolicited, similar to an interrupt; a pull event must be explicitly requested by the consumer, which is similar to polling. Unlike remote procedure calls, which simply fail if a connection is broken, events are stored when they cannot be delivered, and are forwarded when the connection is reestablished. Moreover, if a single supplier is connected with multiple consumers, only a single event needs to be supplied, and all consumers will eventually receive the event. With a little additional coding, perhaps 30 lines of Smalltalk, an application that already uses the Smalltalk MVC update mechanism can be made to transmit events on a change notification and interpret the sent events as update notifications. Several examples of this are supplied with DST, and can be used to effect the initial distribution of a existing application. Ultimately, the application should be converted to the SO/PO split, which takes better advantage of the RPC transport to minimize network traffic generated by change notifications.

DESKTOP LAYER

The topmost layer in the DST system is the HP Desktop. This is a completely distributed graphical application that, although incomplete, serves as an example of the power of a well-distributed application, and provides a



The main Office folder.

reasonable user-interface framework in which to develop such applications. The contents of the Desktop are not part of the CORBA spec, nor are they necessary for the implementation of distributed applications.

The desktop metaphor is a spin-off from the HP NewWave project. NewWave was considerably ahead of its time, both in its pervasive use of objects and its advanced notions of application distribution and reliance on loosely coupled components. The DST desktop uses a building metaphor to refer to other machines on the network and an office metaphor to represent the desktop of a single user. Using reference links, it allows you to place a link to another user's office on your desktop, and, from there, access any of the information in the other user's simulated office. The openness of the office metaphor can be more precisely controlled through the use of login IDs and access control lists (ACLs). For instance, you can allow read-only access to your desktop, and write access to documents on which you are collaborating with your colleagues.

There are several clever applications on the desktop that are genuinely useful tools in their own right. The Forum tool places a shared window on several desktops for participation in a shared discussion. Any object on the desktop can be dropped onto the forum, where it is then graphically rendered on each machine. Each participant is assigned a different color paint, and marks made with anyone's mouse are seen simultaneously on each machine. Coupled with an audio or video hookup, the Forum provides a ready-made groupware facility. Because the source code for everything is shipped with the system, you could take off in many different directions to enhance such a tool.

SUMMARY

Due to the complexity of the CORBA spec, the large volume of the DST classes, and the generally more difficult nature of distributed concurrent programming, the learning curve for DST can be quite steep. HP offers a one-week DST course that assumes a minimal understanding of Smalltalk and object concepts, but, after taking it, I don't see how anyone who has not done significant programming in Smalltalk would get much out of the class. Understanding objects is hard enough; understanding proxy objects and remote message passing seems to require more than mere exposure to Smalltalk concepts.

DST represents to me one side of an important triad of software development technology: Smalltalk, an object database, and distributed computing. Large object-oriented client/server applications are difficult to build without a productive programming language with decent screen-design tools, a persistence system that is closely coupled to the programming language, and a means for efficient communication over a network. The lack of any

continued on page 33

Recruitment Center

JOIN *the* REVOLUTION

For more than 21 years **HBO & Company (HBOC)** has been pioneering the development, delivery and support of fully integrated software solutions for all aspects of the healthcare industry. Growing in excess of 25% a year, our current revenues are over $327 million and reflect our commitment to reach—and exceed—our most ambitious technological goals.

MAJOR DEVELOPMENT CENTERS

Atlanta, GA • Amherst, MA • Minneapolis, MN
Eugene, OR • Salt Lake City, UT • Orlando, FL

We have challenging opportunities for innovative software professionals to analyze, design, develop and implement our highly progressive healthcare information systems. Requires experience in one of the following:

SmallTalk • C++ • Visual Basic
SQL Windows • C/UNIX * Sybase

Your expertise will be rewarded with an exceptional compensation and benefits package. For consideration, forward your resume to: **Corporate Recruiting LHP/SR/0495, HBO & Company, 301 Perimeter Center North, Atlanta, GA 30346. FAX: (404) 393-6063. E-Mail: sharon.hay@hboc.com** EOE M/F/D/V.

HBO&Company

HCm

SMALLTALK DEVELOPERS

HCm, Inc. the leader in providing Decision Support Systems and services to the healthcare industry is seeking several **Smalltalk** developers to join our growing team of professionals. We are an extremely dynamic software development company committed to full-scale OBJECT-ORIENTED development. Coad Yourdon is the methodology of choice with both **Digitalk Smalltalk** and C++ as the implementation OOPL's. If you have a proven history in designing and implementing objects, we'd like to talk to you!

At *HCm*, our values include integrity, professionalism, respect for others, and an enjoyable work environment. If your values match and you are seeking a challenging opportunity to work with a leader, send your resume and salary history in confidence to: W. Buchanan, *HCm* Inc., 3655 Torrance Blvd., Torrance, CA 90503. FAX (310) 316-3781. EOE

COLORADO CAREER OPPORTUNITIES!

Located at the foot of the Rocky Mountains outside of Denver, Colorado, Antalys is a growing and progressive software development and consulting company where creative and responsible individuals thrive. We are continually searching for qualified object oriented developers, designers and architects.

We would like to talk to you if you have experience in the following areas:

- Smalltalk
- C++
- OOA/OOD

Most positions require travel.

For immediate consideration, please send your resume in confidence to:

Antalys, Inc.
1697 Cole Boulevard, Suite 100
Golden, CO 80401
Fax: (303) 274-3030

We are an equal opportunity employer.

Antalys

SEE OUR ANNOUNCEMENT AS CERTIFIED PARCPLACE CONSULTANTS.

Smalltalk RothWell Smalltalk RothWell

SMALLTALK PROFESSIONALS

This is your opportunity to join the finest team of Smalltalk professionals in the country!

RothWell International has challenging projects across the US and abroad.

Excellent compensation and immediate participation in the Employee Stock Plan.

ROTHWELL™ International

BOX 270566 Houston TX 77277
(713) 660-8080;Fax (713) 661-1156
(800) 256-9712; landrew@rwi.com

Smalltalk RothWell Smalltalk RothWell

RothWell Smalltalk RothWell Smalltalk RothWell Smalltalk RothWell Smalltalk

continued from page 31

of these three technologies makes any client/server application unnecessarily difficult to build. VisualWorks provides the rapid development language and reasonable, portable screen design, but no persistence and no communication facilities; an object database management system (ODBMS) such as GemStone provides object storage tightly coupled to the language itself, but has no simple and efficient means of notifying client applications of changes in the database state; DST effects rapid, dependable communication among clients, but no persistence. Using these three key technologies, a Smalltalk client can store information for another client in the database, and, using DST, notify the relevant client of the update. Because the transmission of large objects over the network is not very efficient, the database manages the voluminous information while the network serves merely to notify clients. Using store and forward events, the application can continue to function even in the presence of network faults. I cannot imagine many applications that could not be built using this combination of technologies.

the only way to handle the increasing workload is to distribute it over multiple computers.

CONCLUSIONS

As applications become more and more complex, and the demands for inter-application communication become more pronounced, distributed computing is clearly the next big thing. As machines become cheaper and more interconnected, and as we reach the physical limits of computing technology, the only way to handle the increasing workload is to distribute it over multiple computers. DST is an elegant solution to many of the problems of distributed computation and, as the CORBA spec becomes more widely implemented, DST will enable the distribution of a wide variety of cross-platform and language-independent applications.

Jim Haungs is the founder of TeamTools, Inc. He specializes in Smalltalk consulting, training, project management and software development. He has a BSCS from RIT, an MSE degree from Wang Institute, and is an HP-certified DST consultant. Jim lives in Boston, and can be reached at jhaungs@teamtools.com.

Smalltalk Solutions '95 was a coming of age

David Carr

David W. Curry came to Smalltalk Solutions '95 knowing he wanted to recommend Smalltalk to his bosses at Entergy Services Inc. Although the official decision remained to be made, a trip to OOPSLA had convinced him of Smalltalk's value. Curry was one of many who saw Smalltalk Solutions '95 as an opportunity to find out more. Held Feb. 21–24 in New York City, the conference was the first large gathering devoted solely to Smalltalk. In particular, Curry was seeking to close in on the choice of a vendor. He returned to New Orleans having made all the right contacts. "To me, every minute of this conference was valuable," Curry said.

Entergy, which owns a string of Southern electric utilities, stands to be a major new customer for Smalltalk. Curry made sure to show the vendors his company's position on the INFORMATION WEEK 500 (number 218) and the program from the company's annual Information Technology Conference, which he said is larger than the entire Smalltalk Solutions show.

Curry's experience typifies a trend that Smalltalk Solutions '95 made obvious—the coming of age of the "One True" object-oriented language after almost 25 years. Suddenly, it seems that every major corporation has at least a pilot program for Smalltalk development. Ray Wells, the director of IBM's Object Technology Practice, testified that many corporations are going far beyond that. "We are finding more, and more, and more that major corporations are using Smalltalk and betting the business on it. That takes a lot of guts," Wells said.

They are doing it because Smalltalk programmers can produce fast results instead of the usual string of excuses. "When the businesses come in and say, 'can you?' we say, 'certainly!' When they say 'how long?' we say, 'when do you need it?' We are the implementors of change, not the impediments to change," he said.

Still, it takes guts because major problems remain unsolved—prime among them a drastic shortage of trained and experienced Smalltalkers.

It's supposed to be nearly impossible to write procedural code in Smalltalk, but the untrained and inexperienced somehow find a way, Wells said. "We find that 60% of the Smalltalk we look at is really COBOL—hard to believe, isn't it?"

Yet the mere existence of a large conference devoted to Smalltalk—with attendance running double what the organizers expected—was a promising sign to many.

"To me, this is a historic moment," said William Woo, who is in charge of the Distributed Smalltalk project at Hewlett-Packard. "This is really a healthy start-up." Woo didn't give a presentation because, until the last moment, he didn't expect to be able to attend. He came because he wanted to meet other movers and shakers. "Most of the key players I'm aware of have some representation here," he said.

Trevor Hopkins said he knew Smalltalk Solutions wasn't the first industrial Smalltalk conference. He organized one himself when he was on the faculty of the University of Manchester, although it only drew 70 or 80 people. "I think it's fair to say this is a significant development. It's got a little more international flavor, and it's certainly on a much larger scale. It's clear SIGS and the vendors pulled out quite a few stops," he said.

As a new member of IBM OTP, Hopkins said he used the conference as "a good opportunity to impress on some of these damn Americans that there is a fair amount of Smalltalk expertise on the other side of the Atlantic."

Object Technology International (OTI) President David Thomas said the conference turnout didn't surprise him at all, because he has seen how fast his client list is growing. "Smalltalk is the best-kept secret. Everybody is using it. All you have to do is look at the job market. You just can't hire people who know it, and that's been true for the last three years."

Thomas, who got equal billing with Wells and Object Design International (ODI) President Thomas Atwood as a keynote speaker, argued that Smalltalk is capable of solving every challenge faced by large-scale object development. He has demonstrated it running on a mainframe, plans to have a PDA version by the end of the year, and envisions that by the end of the 1990s it could be the basis for a pure O-O distributed operating system that would dispense with all the communication protocol hassles of contemporary systems. "Folks, we made it to Main Street. We have an obligation now to make it useful to others," he said.

Atwood said objects are moving onto the same growth

David Carr is Manager of Editorial Services at Digital Communications Services, which provides a variety of technical communications services, including documentation of Smalltalk frameworks. He can be reached at davedcs@pcnet.com.

SMALLTALK SOLUTIONS '95

"Excellent — great first conference..."

Bob Moore, President, MCS Data Services, Palm Bay, FL

...was the overwhelming consensus of the 1,000 Smalltalk enthusiasts who attended **Smalltalk Solutions '95** in New York City in February. Attracting an international audience of Smalltalk programmers, developers, technical managers, and consultants, this event marked the first large-scale, vendor-independent conference and exhibition devoted exclusively to Smalltalk.

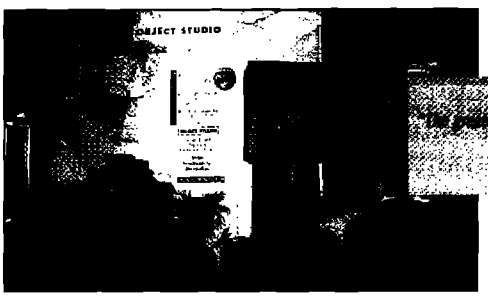
Registered delegates attended classes taught by such Smalltalk leaders and innovators as **Kent Beck, Rebecca Wirfs-Brock, Kenny Rubin, John Pugh, Wilf LaLonde,** and **Sam Adams.** Offering over 30 technical classes, panel discussions, and hands-on workshops, **Smalltalk Solutions** delivered training for all levels of Smalltalk users. Case studies were also incorporated directly into the program, bringing classroom theory to life.

"You draw experts in the field. This is vital for a good conference... I will

—Brad Bowden, Systems Engineer, National Bank, Dallas, TX

In the Exhibit Hall, Smalltalk vendors provided a view of the future of the language. Smalltalk users making serious purchasing decisions could demonstrate and compare different Smalltalk products first-hand, and have questions answered personally by knowledgeable representatives. Useful product information and training was gained through in-depth Product Education Sessions from Knowledge Systems, IBM, Digitalk, Easel, Mark Winter, The Object People, and QKS, as well as a Technology Briefing on NEXTSTEP from NeXT Inc.

Attendees took advantage of other special events throughout the week, including the NY Smalltalk Users Group meeting and other peer group discussions, informal walk-in clinics with the speakers, and keynote presentations by **Dave Thomas, Thomas Atwood,** and **Ray Wells** on present and future uses of Smalltalk.



"The greatest career move since I changed from C++."

—Joseph Pelrine, Consultant, Basel, Switzerland

The premiere of **Smalltalk Solutions '95** was an exciting and educational event for anyone involved in the fast-growing Smalltalk community. Be sure to mark your calendar for next year's event:

Smalltalk Solutions '96
March 4–7, 1996
NY Marriott Marquis, New York, NY

To be sure you receive the most up-to-date conference information, please contact the Conference Registrar at:

SIGS CONFERENCES
71 West 23rd Street
New York, NY 10010

Phone – 212.242.7515
Fax – 212.242.7578
email – info@sigs.com

or check the SIGS Home Page for the latest offerings from SIGS Conferences, SIGS Publications, and SIGS Books:

WWW – http://www.sigs.com

curve that relational database technology followed, with analysts "projecting some pretty good numbers by the end of the decade—enough to attract a lot of venture capital into this market."

Object databases like his company's Object Store are also moving into primetime. That will simplify the lives of Smalltalk developers by eliminating the need to map objects to relational database tables. That conversion makes relational databases two to three orders of magnitude slower at operations involving objects, Atwood said. With object databases, he said, "Objects in Smalltalk are objects in the database—there is no translation. Access to objects in the database is nearly as fast as access to objects in Smalltalk memory."

Better yet, the methods for querying objects in the database are the same as the methods for querying objects in memory. Atwood contrasted that to the relational approach by showing a screen of cryptic SQL next to one sentence-like line of Smalltalk. "This, my mother could read. That difference is worth tens of millions of dollars to large organizations," he said.

One of the conference's case studies echoed the growing acceptance of object databases, albeit in praise of one of ODI's competitors. Texas Instruments (TI) Fellow John McGehee said his team felt it had no choice but to chose an object database when it selected Servio's Gemstone. McGehee acknowledged that object databases are widely considered to be unproved on a large scale. "We were leery, too, and we're going way out on a limb. But we have not seen one area where Gemstone as a product or a company has not come through," he said.

As part of a larger mechanical process reengineering effort, TI built a set of Smalltalk frameworks designed to move chips through its semiconductor factories more efficiently. The language allowed McGehee's team to finish its pilot project ahead of schedule and under budget. One of the frameworks proved so flexible TI is now selling it to other semiconductor manufacturers as ControlWORKS. Commercialization of other frameworks is also planned, McGehee said. "We like to think we're moving in the direction of making every application in the building a framework," he said.

For all the good news, any number of unsolved problems cropped up repeatedly in the course of the conference. For instance, the garbage collection mechanism for removing unused objects from memory—thereby eliminating the memory management problems that tend to be so burdensome in C++—is one of the most frequently cited advantages of Smalltalk. Yet its basic assumption that most objects expire shortly after they are created means objects that make it into long-term memory don't get the same kind of close scrutiny. This can be disastrous when objects survive just long enough to evade the garbage collector.

Atwood said every variety of Smalltalk fails this test "in the face of consistent, high-volume database use." The same issue dominated Kent Beck's lecture on "Building High-Performance Smalltalk Applications," with Beck offering advice on how to tune the garbage collector to control the number of objects that make it into "tenured" memory.

The more controversial aspect of Beck's talk centered on his argument against the supremacy of object diagramming techniques. Beck said he always suspects that "people who write stacks and stacks of diagrams are afraid to program." Those who fail to make use of rapid prototyping are forsaking one of Smalltalk's greatest strengths, he said. Using that power is like drawing on a line of credit, which is longer in Smalltalk than in other languages. "I call this the 'Smalltalk gives you more rope' phenomenon," he said. Just as a business incurs interest costs when it draws on a line of credit, rapid development is not without costs, Beck said. The trick is to remain alert for the point where the cost becomes too high, he said.

"A lot of performance tuning really comes down to confidence," Beck said. "First, get the program running, figure out what the structure ought to be. Once you get the structure right, you can fix the performance problems. Every object becomes a point where you can tweak your screwdriver and improve performance."

Several audience members challenged Beck's approach of designing on the fly as part of development, which they said wouldn't work in systems requiring distribution or concurrency. Beck said he didn't want to minimize that problem. "But my response to that is not to sit down and draw bubbles and arrows but to develop a system."

Anyone who fails to get their hands on such a major problem in the first week ought to be fired, Beck said. "I do have a CASE tool, and I call it Smalltalk. It works better than any specification language I know."

Ted McKnight, president of the New York Smalltalk Users Group, said the conference was very effective at "introducing a lot of people to a subject that they'd been considering but didn't know how to get into." Many of the attendees he met were still preparing to take the plunge, he said. "Most of them are probably departing with more questions than they started with."

"People have a lot of questions about Smalltalk," agreed Terry Montlick, a consultant from Bethlehem, CT. "They suspect there must be a better way of doing things, and they're right. I was first exposed to Smalltalk a dozen years ago, and I have to admit I didn't get it then." Only after trying C++ did he appreciate Smalltalk as a no-compromise O-O environment, he said.

Meanwhile, Curry felt he was going home with a fair number of answers, although he expects several of the vendors to pay a follow-up visit to New Orleans before he makes a recommendation.

Will Entergy bet its business on Smalltalk? "I don't want to say that because that's not the official goal," Curry said. But he is gearing up for the proof-of-concept—and a chance at bigger and better things for Smalltalk.