THE Smalltalk REPORT
PROVIDING SOLUTIONS TO THE SMALLTALK COMMUNITY

Table of Contents

September 1995 Vol 5 No 1

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

When ParcPlace announced their proposed merger with Digitalk, devotees of VisualWorks and Visual Smalltalk were left with many unanswered questions. Would both dialects remain intact? Would one dialect be dropped in favor of the other and, if so, for how long would the discarded dialect continue to be supported? Would a new dialect emerge and, if so, would the base for the new Smalltalk be VisualWorks or Visual Smalltalk? What would happen in the area of team programming and configuration management? Would the combined company take up the challenge issued by the Gartner Group, who urged Smalltalk developers to challenge the two companies to commit to a coherent strategy for the integration of the libraries, syntax, and tools of the two dialects? How would the investment made by organizations who have adopted one dialect or the other be protected? It was no surprise, therefore, that the close to 1,000 Smalltalkers attending the ParcPlace-Digitalk International Users Conference awaited the opening presentations with much anticipation.

The combined company will have the technical and professional services personnel to compete with the competition

Digitalk had been courted by a number of companies but if we examine the strengths and weaknesses of the two companies and their Smalltalk dialects, the rationale for the merger with ParcPlace and the future path to be charted by the combined company becomes evident. Digitalk's strengths lie in their visual programming, component, and packaging technology (PARTS Workbench, SLLs), and platform look and feel (Windows and OS/2). ParcPlace, on the other hand, brought to the table a much wider range of platforms including UNIX (a major weakness of Digitalk), better support for new application creation and enterprise-wide computing, "image-level" portability across platforms, and, arguably, better access to relational and object-oriented databases. As summarized by PP-D, the strengths of ParcPlace and Digitalk are in "producer" programming and "consumer" programming respectively. There is also a question of size that comes into play. The combined company will have the technical and professional services personnel to compete with the competition from the likes of Powersoft (PowerBuilder), Microsoft (Visual Basic), Forte, and, dare we say, IBM.

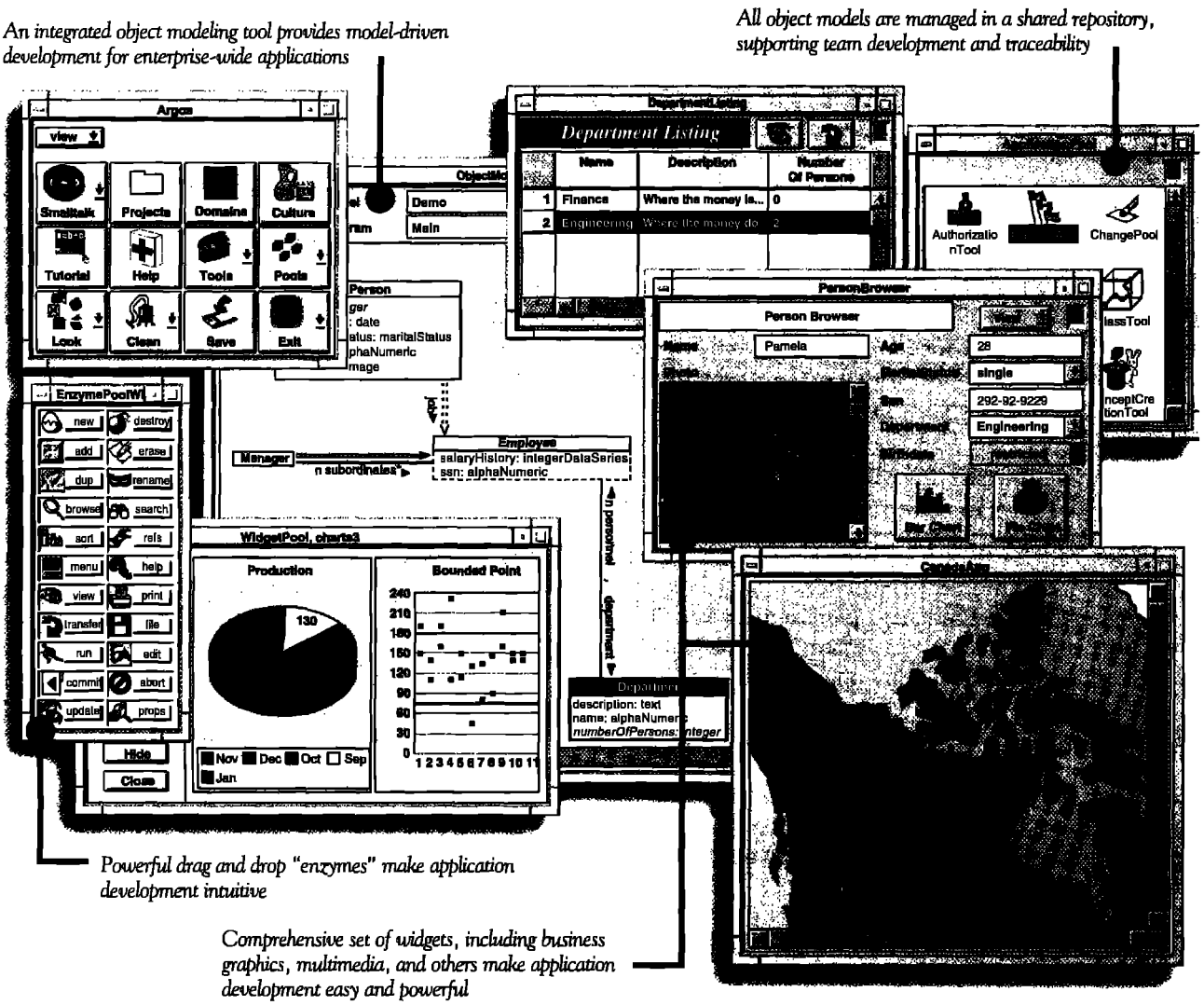
The strategy outlined by PP-D at the conference was to release new versions of both products (VisualWorks 2.5 and Visual Smalltalk 3.1) as planned in 4Q 1995 but to work toward a converged image to be released in 1996. The latter, known internally as "Van Gogh," would be based on the VisualWorks engine with the addition of key capabilities from Visual Smalltalk such as component assembly (via "wiring"), component packaging (SLLs will replace the now interim notion of parcels in VW 2.5), Team/V for team programming and configuration management, platform compliance via host integration, and the event mechanism. There will be no further development on Visual Smalltalk after Release 3.1.

A look at the features in the upcoming releases of both products plus the long list of features planned for the combined image again emphasizes why the companies found the merger an irresistible proposition. OLE support, better performance, headless capability, certification (Windows '95 for Digitalk and Windows NT 3.51 for ParcPlace), and compatability with the ANSI standard are to be found in both of the new releases. In the future, PP-D promise superior clients and servers, much enhanced support for distributed computing, and tolls for network and application management. Dedicated clients will have improved platform compliance and better interoperability with other languages; they will be faster and take up a smaller footprint. Application servers will be tuned for deployment of shared use of business model components and provide outstanding throughput, concurrency, scalability, and reliability. This is an awesome list. If they succeed in the aggressive timeframe they have set for themselves, the merger will be more than vindicated.

The highlight of the conference? Not much doubt about that. The most captivating technology and the most compelling presentation involved ParcPlace's new World Wide Web technology. Michael Robicheaux, better known as Roby, appeared on stage on a Harley Davidson piloted by no less than Adele Goldberg in full biker gear. In a carefully staged event, Roby demonstrated how a VisualWorks windowSpec could be rendered as HTML rather than Windows or Motif, by showing a Netscape client attached to a VisualWorks server. The "coup de grace" was to see Roby bring up a Netscape Smalltalk browser page to fix a bug in a Smalltalk application directly from NetScape! Wadsworth, as the WWW technology is called, will appear next year.

Introducing Argos

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



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

Managing asynchronous network messages from external applications

Michael Christiansen

RECENTLY, WE WERE ASKED TO DEVELOP a service that allows applications written in ParcPlace VisualWorks to respond to messages received from external sources: support applications written in C++ and attached to telecommunication switches and other components of a cellular phone network. Overall, this effort (the Cellular Performance and Management System, or CPMS) is to provide performance monitoring and control facilities for these networks and their individual components.

Messages from these external sources are delivered via inter-processor communication services provided by the underlying operating system, e.g., pipes and sockets. The information is delivered to our system in an asynchronous manner in the form of event and error notifications. We were required to respond to the asynchronous delivery of messages from the network elements, extract and process the information contained in the messages, and perform one or more operations with the information. These messages are described as asynchronous because we cannot predict when or how often they will be delivered to our applications.

During the initial phases of CPMS, the issues related to inter-process communications (IPC) and communications with the network elements were analyzed. We defined what were felt to be important goals for the design and implementation of the services described in this article. The highlights of these goals are:

- **Implement CPMS IPC in a separate application.** Our interface to the telecommunications network components was accomplished through a C-based API we didn't wish to integrate into the Smalltalk object engine for the following reasons. First, we felt the API might comprise the integrity of the object engine in ways that were hard to detect and debug. Also, the API employs a callback mechanism, similar to the X-Windows System callback, which would have been difficult to integrate with the object engine. Finally, API integration would have severely constrained our application's portability.
- **Apply an interrupt-driven mechanism to detecting delivered messages.** This can be contrasted with the decision to apply a polling mechanism where the receiver periodically polls the communications channel, checking if any messages have been received. Polling wastes processing cycles if the polling rate is too high, or causes unnecessary and possibly dangerous delays in the response to a received message if the polling rate is too low.
- **Integrate our message delivery notification mechanism with Smalltalk's Model-View-Controller (MVC)** and the underlying framework that permeates Smalltalk's class library. Specifically, we wanted to provide model-like behavior, where an ICP channel's dependents would be notified via the traditional update: protocols when a message arrives.
- **Instance-specific behavior to define the processing of messages.** We wanted the option of allowing for instance-specific behavior when interpreting the messages that arrives from a network element. This is contrasted with constructing a subclass for each type of message and embedding the specific behavior in the subclass. The subclassing approach would cause a proliferation of classes that had only a single instance.

This article describes a ValueModel we developed to meet these goals. But beyond a discussion of how this was accomplished, this article presents solutions to several of the pitfalls and problems present in the development of any mechanism that applies socket-based IPC in a multithreaded environment. The implementation of the solution is available in the VisualWorks sections of the Manchester Goodies Library as ExternalReadStreamAdaptor.st.

DESIGN

This section describes the design of ExternalReadStreamAdaptor (ERSA), a ValueModel subclass that allows its users to construct and install a process that listens for and decodes messages received through an ExternalReadStream.

Like the ValueModel, an installed instance of ERSA behaves as a model whose dependents are notified when its value changes. The value of the ERSA is set according to the information provided by the messages it receives and decodes.

The behavior that defines the decoding of a received message is provided by the user of the ERSA. During instance initialization, the user provides two blocks that are evaluated for every received message. The evaluation of the first of these blocks reads the message off the external stream. The second block produces the new value of the ERSA instance.

The processing that takes place for every message received is illustrated in Figure 1.

The ERSA instance is initialized with an ExternalReadStream that has been attached to a communications channel, for example, on an open socket connection. Message handling is then started within its own processing context. The process first waits for a message to arrive. Once information is detected on the stream, the information is read and decoded to create a new value for the ERSA. Finally, the dependents are notified via the changed update protocol inherited from ValueModel.

The processing described above is executed within an independent process (thread) within the image. This feature is necessary because messages are received asynchronously from external sources, independent of the processing state of the applications and other objects that make use of the received information. The dependency mechanism provided by the ValueModel allows these applications to register themselves as dependents of an ERSA and receive notification when new information arrives. However, these dependent objects must be aware that the notifications will arrive asynchronously and take whatever precautions necessary to ensure that their states are properly maintained.

Because each instance of an ERSA maintains its own process, or thread, it is important that each thread is properly scheduled. That is, each ERSA's thread should wait in a suspended state until a message arrives on its external stream. The mechanism used to schedule the thread is provided by the IOAccessor maintained by the ExternalConnection, which in turn is maintained by the ExternalReadStream. The method IOAccessor>>read-Wait will suspend the thread that makes the call if no data is available on the IOAccessor's external connection. This external connection is usually a socket or pipe, and is provided by the underlying operating system. This service of the IOAccessor allows each instance of an ERSA to wait in a suspended state until information arrives for it across it's external connection. At this point, processing and dependent notification occurs, after which the adaptor again waits in a suspended state for the next message to arrive.

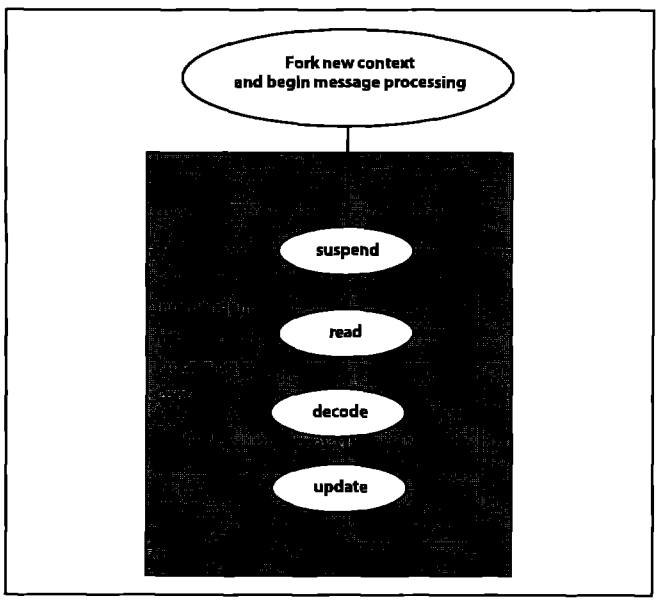


Figure 1. Messaging processing loop.

Message processing

Each ERSA instance receives messages from its peer external process across the IPC channel they share. In its simplest form, this message can be seen as a stream of bytes to be interpreted and converted into a value useful to the adaptor's dependent objects. The ERSA instance must be prepared to decode the received information into the type of value expected by the application. For example, length of a queue or capacity measurement from a cellular switch might be transmitted by the external process in an ASCII format. The adaptor would be required to decode this ASCII information into an integer, float, or higher-level abstraction such as a running average of the values.

An important aspect of defining the message format is the mechanism used to detect the end-of-message on the receiving end, i.e., the ERSA end of the connection. An ExternalStream provides no mechanism for determining when an entire message has been read. In fact, attempting to read a byte from an empty ExternalStream will cause the calling thread to suspend until data arrives.

Understanding the format of the message that will be received from the external process is the responsibility of the ERSA instance. The developer first determines the format and end-of-message marker to be utilized by the external process, and then encodes this format into the reader block used to initialize the ERSA instance. This process is described in more detail in the section describing implementation issues.

Message encoding and decoding. Because the peer external process is implemented in C++, there exist limited options in the methods employed to encode its messages. We found that either we could transmit the information in its native binary representation or encode the binary information into ASCII strings.

A binary representation would require both the ability to convert native binary data to Smalltalk objects, and to detect and translate differences in the native binary formats for the sending and receiving processors. Methods of translating binary representation into Smalltalk objects and for marshaling between native binary formats is provided by the class UninterpretedBytes.

The alternative to binary representation is to encode and decode the binary values to and from ASCII strings. C++ applications can implement the encoding of a binary value into ASCII using the function sprintf or an equivalent stream operation.[1] The examples presented in this article demonstrate ASCII decoding techniques in Smalltalk.

Within Smalltalk, there exist several options for decoding the received ASCII strings. All the options described in this article employ the Smalltalk compiler to translate expressions into object instances. An alternative is to

develop an application-specific translator using the compiler construction tools provided by ParcPlace's advanced programming tool kit.[2]

In the simplest of implementations, the compiler can be used to evaluate the message. For example, try evaluating the following code:

```
wrstr := String new writeStream.
101 storeOn: wrstr.
Compiler evaluate: wrstr contents
```

The code segment above causes the compiler to evaluate the contents of the stream and returns a reference to the object 101. In the example, the integer value is encoded via the storeOn: method.

Note that there are some important differences between the example given above and an acceptable ERSA implementation where the stream being processed will be an ExternalReadStream. It is not safe to use the contents message to extract the message from an ExternalStream, because the contents message relies on state information not available to an ExternalStream. The caller has no way of knowing if the entire message has been retrieved from the external connection by a single contents operation. The following section discusses the issues and options related to detecting the end of a received message.

The previous example demonstrated the simplicity of producing a Smalltalk value from an ASCII encoding. However, the compiler can only translate valid Smalltalk expressions, and this approach to decoding fails when complex values need to be encoded and decoded. For example, when two values are transmitted, the following code segment produces a compiler exception:

```
wrstr := String new writeStream.
101 storeOn: wrstr.
wrstr nextPut: Character space.
232 storeOn: wrstr.
Compiler evaluate: wrstr contents
```

This compiler error occurs because the contents of the stream "101 232" is not a valid Smalltalk expression. The sender would need to embed the values in an expression generating a container such as #(101 202).

But there is an alternative to the compiler: applying the Smalltalk Scanner to tokenize the stream of characters received from the external stream. For example:

```
wrstr := String new writeStream.
101 storeOn: wrstr.
wrstr nextPut: Character space.

232.94 storeOn: wrstr.
Scanner new scanTokens: wrstr contents.
```

This code segment produces an array of two values: an integer and a float. The scanner will also decode text and single quoted strings.

The above examples demonstrated how a simple encoding of values into ASCII representation can be

decoded by a Smalltalk application. But if the sending process has been implemented in Smalltalk, additional encoding options exist such as the use of storeOn: (as in the above examples), or the use of BOSS encoding and decoding as described in White et al.[3]

End of message detection. The second issue in defining the message protocol between the sending process and receiving ERSA instance is the end-of-message marker used to locate the last byte of the message. In one respect, the marker is needed to determine the separation of two messages queued up in the IPC connection. But, more importantly, the marker is used by the ERSA instance to ensure no attempt is made to read on an empty external stream.

The protocol defined between the sending process and ERSA reader must ensure that the reader does not attempt to read past the end of a message. As stated earlier, if an attempt is made to read from an empty external stream, the thread performing the read operation will suspend. This makes it impossible for the ERSA reader to decode the message and update the dependents directly upon the receipt of the message.

The protocol defined between the sending process and ERSA reader must ensure that the reader does not attempt to read past the end of a message.

There are at least two approaches to determining the position of the end of a message in an external stream. One is to prepend the length of the message to the head of the message itself. The reading process can read that value and then extract only that number of bytes from the external stream. The Scanner is useful in reading and decoding the numeric value from the head of the message.

A second approach is to append a special character (any character value not used in the encoding of the message) to the end of the message. The reader process can extract characters from the stream until the special character is found. In the case of ASCII text, the carriage return (cr) is often applied as such a special character. In the UNIX environment, control-d (integer 4) is often used to denote the end of a file or message.

The Smalltalk Stream class offers support for employing a special character as an end-of-message marker. The method Stream>>through: returns the contents of the target stream, starting at its current position through the position of the first instance of the argument, e.g.:

```
anExternalStream though: Character cr.
```

This code segment returns a Collection subclass with the contents of the ExternalStream from its initial position up to and including the position where the cr was located.

Exceptional conditions

Figure 1 illustrates the processing of messages arriving from the external process under ideal conditions. However, there are exceptional conditions that can occur, and are the responsibility of the ERSA to detect and recover from.

The first exceptional condition occurs when the external process closes its end of the IPC channel, e.g., when it finishes processing or if the external process unexpectedly crashes. The closed connection must be detected and hooks provided to allow the developer to respond accordingly.

The second exceptional condition arises when the ERSA reader receives a partial message due to errors in its transmission across the IPC channel or errors in the sending external process itself. The partial message will cause the ERSA instance reading the message to suspend, waiting for the full message to arrive. Again, it is the responsibility of the adaptor to detect this condition and supply the developer the hooks needed to recover.

Detecting when the IPC channels has been closed is performed by the IOAccessors class and is integrated into the design of ERSA. This is described in the section covering implementation issues.

Detecting an incomplete message is complicated by the fact that ExternalStream does not provide a mechanism for "read with time-out," i.e., read the next byte from the target stream or time-out if no data is available. If the external process sending the message across the IPC connection fails to follow the agreed upon protocol between sender and receiver, the ERSA reader will not detect the end of the message and will continue to read bytes off the stream until it is empty with the unfortunate results described earlier.

Our solution to the lack of a read with time-out is to create the service. This is accomplished by nesting the acquisition of the message in its own thread, and raising a time-out signal if the message has not been acquired within some predefined interval. This is illustrated in the following code segment:

```
localPDelay := Delay forSeconds: 5.
messageOK := false.
localProcess :=
    [theMessage := self readMessageFrom: aStream.
    messageOK := true.
    localPDelay delaySemaphore signal] fork.

localPDelay wait.
localProcess terminate.
messageOK
    ifTrue: [self value: (self performProcessingOn: theMessage)]
    ifFalse: ["Raise a messageAcquisitionTimeoutSignal"]
```

This example demonstrates how the acquisition of the message is performed within an independent thread whose parent context maintains a time-out delay. Once the delay semaphore is released, the parent context will signal an error if the messageOK flag has not been set true.

Within the context referenced by localProcess, the delay semaphore will be signaled (released) immediately after message processing has successfully completed, minimizing the amount of delay in the notification of dependent objects.

Also notice the call to "self value:" that set the value of this ValueModel subclass to the value returned by the message's processing and notifies dependent objects of the change.

Message processing exceptions. During the processing of the message, any form of exceptional condition might arise, e.g., arithmetic errors, etc. Because the processing of the message is performed within its own process context (see Fig. 1), the parent context cannot set up handlers to be caught by errors that occur within the child (ERSA instance) context. More specifically, a signal raised in a child context will not be caught by an exception handler installed by the parent context.

To address the situation, each instance of an ERSA can be initialized with and maintains an instance of a HandlerCollection. The handlers maintained by this collection are installed within the message processing context to catch exceptions that arise during the processing of messages.

IMPLEMENTATION

The design issues described in the previous section were implemented in the class ExternalReadStreamAdaptor. This class provides instance-based configuration of the behaviors needed to read and process messages, as well as handlers for exceptional conditions.

While the class has been designed to be configured for instance-specific behavior, its instance and class methods have been designed to allow for subclassing to define application-specific behavior.

While this section describes the implementation of the services provided by the class, we felt that including the source code in the article would either constrain the amount of detail given to the specific issues we encountered, or would make the article too long to be publishable. Instead, the source code of the classes can be acquired from the Manchester Goodies Library, or the reader can send a request to the author at the addresses appearing at the end of the article.

Instance initialization

An instance of the ERSA is initialized with:

- the ExternalReadStream, which provides the connection to the peer application.
- a block that implements the reading of the message from the external stream.
- a block that decodes the contents of the message and provides a new value.
- optionally, a set of exception handlers to catch signals raised by the ERSA during its processing.

We felt that it should be the responsibility of the user to open and create a ReadStream connection to the peer

application. An example of doing this is given in the following section, and further examples are given in the distributed source code.

There are several reasons for making the management of the external connection the responsibility of the user. First, there are issues related to the different types of external connections that can be created via the socket interface provided by ParcPlace. Second, the opening, closing, and overall management of an external connection can be highly application specific. Finally, external connections are not preserved across image snapshots and so must be reinitialized when the application is started.

The block used to define the behavior of reading the message from the external stream is called the readerBlock. This block evaluated with a single argument, the stream defined during the initialization. The block is expected to return the message it extracts from the stream. For example:

```
[ :anExStream | anExStream through: Character cr].
```

This block returns a Collection containing the data read from the external stream up to and including a carriage return.

The second block, called the actionBlock, defines the behavior that takes the message returned from the above block and produces the value the ERSA instance assumes and provides to its dependents. The actionBlock is called by a single argument, the message (value) returned by the readerBlock. For example:

```
[ :message | Compiler evaluate: message]
```

This block evaluates the contents of the stream returned by the readerBlock. The value produced by the evaluation becomes the current value of of the ERSA instance.

At this point, the reader might wonder why we split the behavior of the reading and decoding the of the message into two blocks when a single block could perform both services. Our reason for this separation was to allow the detection of a message read time-outs independent of the time needed to decode the message.

The optional fourth argument that ERSA accepts is a set of exception handlers encoded into an instance of a HandlerCollection.[4] The application of this set of handlers is described next.

Exception handling

The user of the ERSA is able to provide a set of exception handlers to be installed within the context of the process that the instance maintains. The method readLoop defined in the following code segment illustrates the definition of that process and the processing loop that manages the reading and decoding of messages from the external stream:

```
readLoop

    readLoopProcess := [
        self handlerCollection handleDo: [
            "if the peer connection has been closed the wait-
             ForReadableData will return false. This would
             indicate a closed external stream or other error."
        [self waitForReadableDataOn: readStream]
            whileTrue: [self processAndUpdate: readStream].

        "This branch will be entered when the peer
         connection has been closed."
        ExternalReadStreamAdaptor
            streamClosedSignal raiseWith: readStream.
        self shutdown]] fork
```

The HandlerCollection should provide handlers for those signals that might be raised during the evaluation of the actionBlock (evaluated within the processAndUpdate: method) and two ERSA-specific exceptions. These ERSA-specific exceptions are:

- MessageAcquisitionTimeoutSignal: The acquisition of the message from the external stream took longer than the value set by the class variable MessageReceiveTimeout.
- StreamClosedSignal: That a close (or some other error) was detected on the external connection.

The implementation of detecting and raising the message acquisition time-out was described in the previous section.

Detecting data and closed streams

The method readLoop, described above, maintains a while loop whose test determines the life of the process it is embedded within. The method implementing this test is shown here:

```
waitForReadableDataOn: anExternalReadStream
    anExternalReadStream ioConnection input readWait.
    ^anExternalReadStream atEnd not.
```

This method utilizes two services provided by ExternalReadStream. The first line implements a call to IOAccessor>>readWait. This method will suspend the calling process until data is available for reading from the external connection. The second line of the method implements the test for a closed stream. If the stream has been closed from the peer end, the readWait method will return when no data is present on the external connection. This fact is detected and returned as the value of this method.

EXAMPLE

This section provides a detailed example of the application of the ExternalReadStreamAdaptor. The example includes opening a socket and initializing the ERSA instance.

```
"Create a socket on the given host and port."
socket := UnixSocketAccessor
        new TCPclientToHost: 'localhost'
        port: port.
```

A second approach utilizes a platform and OS-independent method of creating the socket that has been made available in VisualWorks 2.0:

```
socket := IOAccessor defaultForIPC
```

```
        new TCPclientToHost: 'localhost'
        port: port
```

This segment creates a new socket that is connected to the given host and port number address:

```
extCon := ExternalReadStreamAdaptor
    newWithStream: (socket readStream)
    readerBolck: [ readStream |
            readStream through: Character cr]
    actionBlock: [ :charArray |
            Compiler evaluate: charArray]
    handler:
        (HandlerCollection new
            on: ExternalReadStreamAdaptor
                messageAcquisitionTimeoutSignal
            handle: [ :ex | Transcript show: 'processing restart' :cr.
                ex restart];
            yourself).
```

This segment creates a new instance of the adaptor. The ExternalReadStream is created from the socket accessor using the method IOAccessor>>readStream. The readerBlock reads characters from the stream until a carriage return is seen. The actionBlock evaluates that text. The HandlerCollection is installed with a single handler that will restart the processing when raised.

```
extCon readLoop.
```

This method starts the message handling process as described in the previous section.

The distributed source code provides further examples of applying this class, as well as examples of applications that act as the peer connection and GUI applications that display the value of an ERSA instance.

CONCLUSIONS

This article described a-kind-of ValueModel developed to provide a mechanism for receiving and decoding asynchronous messages from external C and C++ applications. The class implementing these services is called the ExternalReadStreamAdaptor.

Rather than demonstrate how peer-to-peer connection is established using socket-based interprocess communications, this article has tried to demonstrate how the MVC pattern can be applied to message decoding and notification of dependent objects. The services provided by this class permit its users to create an independent process that will receive and decode messages received across an ExternalReadStream. The class permits the parameterization of the behavior that reads the expected messages from the socket and of the behavior that decodes the information into the application-specific value.

In the design of the ExternalReadStreamAdaptor, we have attempted to provide a service that addresses the problems commonly found in managing communications across a network channel, e.g., defining and maintaining a protocol between the sending and receiving processes.

continued on page 19

Accessing configuration data in the DOS/MS Windows environment

Dayle Woolston and Bob Capel

THIS ARTICLE ILLUSTRATES some techniques you can use to manipulate DOS/MS Windows configuration data in VisualWorks 2.0 applications. Configuration data for Windows applications are defined in files such as AUTOEXEC.BAT, CONFIG.SYS, WIN.INI and any number of application-specific .INI files. VisualWorks 2.0 applications retrieve and update information contained in these files through the DLL and C Connect product.

The examples in this article were coded on an AST 4/50d running Novell DOS 7.0, MS Windows 3.1, and ParcPlace VisualWorks 2.0 with DLL and C Connect.

WINDOWS KERNEL SERVICES

In the "Appendix: Module and library names" section of THE PROGRAMMER'S REFERENCE, VOLUME 1: OVERVIEW (contained in the Microsoft Windows Software Development Kit), there is a listing of Windows kernel functions, some of which provide features useful for manipulating configuration information.

We define the class WindowsKernel to provide access to the functions found in KERNELL.DLL:

```
ExternalInterface subclass: #WindowsKernel
    includeFiles: 'windows.h '
    includeDirectories: ''
    libraryFiles: 'KERNEL.DLL'
    libraryDirectories: ''
    generateMethods: ''
    beVirtual: false
    optimizationLevel: #debug
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'WindowsKernelDictionary '
    category: 'Windows-ExternalApplications'
```

The DLL and C Connect product includes a tool set that builds function call templates and type definitions such as those which follow. In a future article, we can talk about the utility of these tools. Parsing out dense, hierarchical header files can be a challenge. (Parsing out WINDOWS.H is right on the border of difficult.)

READING THE DOS ENVIRONMENT

Our first example illustrates how to fetch DOS environment text. In this example, we use the function call GetDOSEnvironment, defined in the Windows SDK as:

```
LPSTR _far GetDOSEnvironment(void)
```

We define the corresponding Smalltalk methods on the instance side of WindowsKernel:

```
LPSTR
    <C: typedef char _far * LPSTR>

GetDOSEnvironment
    <C: LPSTR _far GetDOSEnvironment(void)>
    ^self externalAccessFailed
```

The tricky issue here is that Windows passes back a C pointer to a contiguous set of null-terminated strings. The end of the set is identified by the empty string—two adjacent nulls. The following method copies each C string into a write stream and returns a Smalltalk string describing the DOS environment:

```
getDOSEnvironment

    | dstr temp ws |

    dstr := self GetDOSEnvironment.
    ws := (String new: 1000) writeStream.

    [temp := dstr copyCStringFromHeap.
    temp isEmpty]
        whileFalse:
            [ws nextPutAll: temp; cr.
            dstr datum: dstr datum + temp size + 1].

    ^ws contents
```

Inspect (execute) the following code:

```
WindowsKernel new getDOSEnvironment
```

and you will see something similar to:

```
'CONFIG=WORKNET
PROMPT=$p$g
LMOUSE=C:\MOUSE
TEMP=C:\DOS
COMSPEC=C:\COMMAND.COM
PATH=C:\WINDOWS;C:\DOS;C:\MOUSE
```

```
NAME=dwoolsto
windir=C:\WINDOWS'
```

We have found it useful to cache DOS environment information in a dictionary. It does not change within the runtime life of the application and there is no sense in repeatedly parsing out specific entries. The code below allocates a dictionary. Each key gets parsed out as the string preceding the equals character; its corresponding value is the string following the equals character.

```
getDOSEnvironmentDictionary

    "WindowsKernel new getDOSEnvironmentDictionary"

    | dstr dictionary temp |

    dstr := self new GetDOSEnvironment.
    dictionary := Dictionary new.

    [temp := dstr copyCStringFromHeap.
    temp isEmpty]
        whileFalse:
            [ | key value |
            key := temp copyUpTo: $=.
            value := temp copyFrom: key size + 2 to: temp size.
            dictionary at: key put: value.
            dstr datum: dstr datum + temp size + 1].

    ^dictionary
```

Your application may hold on to the dictionary for future reference or you may choose to cache it in WindowsKernel.

In the example above, we created an instance of WindowsKernel to get the DOS environment. Applications that make repeated calls on kernel services should probably create a well-known instance of WindowsKernel. We do this by adding the class variable Default to WindowsKernel. The class-side selector, default, lazy-instantiates its content:

```
default
    Default isNil ifTrue: [ Default := self new ].
    ^Default
```

Create a corresponding getDOSEnvironmentDictionary method on the class side of WindowsKernel and direct it to use its default WindowsKernel instance to return the dictionary:

```
getDOSEnvironmentDictionary
    "WindowsKernel getDOSEnvironmentDictionary"
    ^self default getDOSEnvironmentDictionary
```

READING .INI FILES

Let's next take a look at information available through the Windows environment. Windows configuration information is maintained in the WIN.INI file usually found in C:\WINDOWS. Applications use the kernel calls GetProfileInt and GetProfileString to read information from WIN.INI. They use WriteProfileString to store information in WIN.INI. You may find it convenient to store application-specific information in your own .INI file. Windows provides three similar calls to interact with custom .INI files: GetPrivateProfileInt, GetPrivateProfileString, and WritePrivatePrifileString.

Starting with WIN.INI, we define the following function template for GetProfileString. By convention, the DLL and C Connect tool places this type of method in the instance-side procedures protocol.

```
GetProfileString: arg1 with: arg2 with: arg3 with: arg4
with: arg5
    <C: short _far _pascal GetProfileString(char _far * ,
char _far * , char _far * , char _far * , short)>
    ^self externalAccessFailed
```

For readability reasons, we often prefer to encapsulate the template in another method whose signature is more self-documenting than the with:with:with: convention. In addition, this encapsulation provides the service of allocating a C String buffer required by the call, returning to the caller a conventional Smalltalk string. We place the following method in the instance-side services protocol:

```
getProfileString: sectionString entry: entryString default:
defaultString

    | returnString |

    returnString := CIntegerType unsignedChar gcMalloc16: 80.
    self GetProfileString: (sectionString gcCopyToHeap16)
```

```
        with: (entryString gcCopyToHeap16)
        with: (defaultString gcCopyToHeap16)
        with: returnString
        with: 80.
    ^returnString copyCStringFromHeap
```

Inspect (execute) the following code:

```
WindowsKernel getProfileString: 'windows' entry:
'KeyboardSpeed' default: '0'
```

and you will see something similar to: '31'.

This code works fine for reading single entries from WIN.INI. However, as in the DOS environment dictionary example above, you may want to return the entire collection of entries belonging to a section of the WIN.INI file. Two methods can be useful for this purpose. In the first, we return an ordered collection of section entries. These entries play an identical role to that of the key values described earlier.

```
getProfileSectionEntries: sectionString

    "WindowsKernel getProfileSectionEntries: 'windows' "

    | returnString default oc temp |

    returnString := CIntegerType unsignedChar
gcMalloc16: 80.
    default := '**NONE**'.
    self GetProfileString: (sectionString gcCopyToHeap16)
        with: nil
        with: (default gcCopyToHeap16)
        with: returnString
        with: (80).

    oc := OrderedCollection new.
    [temp := returnString copyCStringFromHeap.
    temp isEmpty]
        whileFalse:
            [oc add: temp.
            returnString datum: returnString datum + temp
                size + 1].

    ^(oc size == 1 and: [oc first = default])
        ifTrue: [OrderedCollection new]
        ifFalse: [oc]
```

Inspect (execute) the code in the method comment and you will get something similar to:

```
'OrderedCollection ('Load' 'NetWarn' 'spooler' 'run' 'Beep'
'NullPort' 'BorderWidth' 'CursorBlinkRate'
'DoubleClickSpeed' 'Programs' 'Documents'
'DeviceNotSelectedTimeout' 'TransmissionRetryTimeout'
'KeyboardDelay' 'KeyboardSpeed' 'ScreenSaveActive'
'ScreenSaveTimeOut' 'CoolSwitch' 'device')'
```

Knowing the entries (keys) to a section, your application can make the appropriate getProfileString:entry:default call to get a value for any specific key. You may want to build a method that returns a dictionary of keys and values for the section—precisely as we did in getDOSEnvironmentDictionary above. Consider the following code:

```
getProfileSectionDictionary: sectionString

    "WindowsKernel getProfileSectionDictionary: 'windows' "

    | dictionary |

    dictionary := Dictionary new.
    (self getProfileSectionEntries: sectionString) do:
        [:entryString |
        dictionary
            at: entryString
            put: (self getProfileString: sectionString
                entry: entryString default: '<none>')].

    ^dictionary
```

Inspect (execute) the code in the method comment and you will get a dictionary whose keys are the contents of the ordered collection in the previous example and whose values are their corresponding strings.

SOME OPTIMIZATIONS WORTH CONSIDERING

The code above works fine. However, with a little additional complexity, we can get some worthwhile optimizations. The gcCopyToHeap16 method is the most general mechanism for

copying data to the C heap and returning the pointer necessary for passing a string through the C interface. However, it has two drawbacks: first, if called relatively frequently, it causes work for the garbage collector, and, second, there is a faster alternative, which we describe below.

Typically, the applications that we write are database-retrieval intensive. Consequently, we have a habit of tormenting the garbage collector through constant SQL selections and object instantiations. We have had to look for more frugal memory allocation strategies. We have adapted one of these strategies to the WindowsKernel class hierarchy described below.

Let's begin by generalizing the WindowsKernel inheritance hierarchy as follows:

```
ExternalInterface
    WindowsInterface
        WindowsDialog
        WindowsKernel
        WindowsUser
```

WindowsKernel retains the same methods (and more) described above. WindowsDialog and WindowsUser implement Windows DIALOG.DLL and USER.DLL calls documented in the MS Windows SDK. They each inherit optimizations implemented in WindowsInterface.

We define WindowsInterface as follows:

```
ExternalInterface subclass: #WindowsInterface
    includeFiles: ''
    includeDirectories: ''
    libraryFiles: ''
    libraryDirectories: ''
    generateMethods: ''
    beVirtual: false
    optimizationLevel: #debug
    instanceVariableNames: ''
    classVariableNames: 'CHeapStrings CHeapStringsIndex '
    poolDictionaries: 'WindowsInterfaceDictionary '
    category: 'Windows-ExternalApplications'
```

The class variables CHeapStrings and CHeapStringsIndex are used by WindowsInterface to manage a round-robin queue of C heap strings. Five methods manage this service. First, the class initialize method for WindowsInterface makes the class a dependent of ObjectMemory. The reason for this will be explained later. Second, we explicitly initialize the class variables to nil. (This is extraneous in a production application, but useful in debugging a development image.)

```
initialize

"WindowsInterface initialize"

    (ObjectMemory dependents includes: self)
        ifFalse: [ ObjectMemory addDependent: self ].
    CHeapStrings := nil.
    CHeapStringsIndex := nil.
    Default := nil.
```


In this rework of the class hierarchy, Default is a class-instance variable defined in WindowsInterface and inherited by WindowsDialog, WindowsKernel, and WindowsUser. In this variable, each class lazy-instantiates a well-known instance of itself.

The C heap strings that we allocate below must be freed up prior to exiting the application or Windows loses subsequent use of these resources. To support this mechanism, we define the following method on the class side of WindowsInterface:

```
update: anAspectSymbol with: aParameter from: aSender

    (aSender == ObjectMemory)
        ifTrue:
            [ (anAspectSymbol == #aboutToQuit )
                ifTrue: [ self release ]
```

Because WindowsInterface is a dependent of ObjectMemory, it gets notified when the image is about to quit, at which time it calls its release method:

```
release

    CHeapStrings notNil
        ifTrue:
            [CHeapStrings do: [:acstr | acstr notNil ifTrue:
    [acstr free]]].
```

cHeapString is a WindowsInterface class-side method that

lazy-instantiates the round-robin queue of C heap strings. In this case, we initialize the queue with 8 strings:

```
cHeapString

    CHeapStrings isNil ifTrue: [
        CHeapStringsIndex := 0.
        CHeapStrings := Array new: 8.
        1 to: 8 do:
            [ :i | CHeapStrings at: i put:
                (CintegerType unsignedChar malloc16:
                    (self defaultLargeBufferSize))]].

    CHeapStringsIndex := CHeapStringsIndex + 1.
    CHeapStringsIndex > 8 ifTrue: [ CHeapStringsIndex := 1 ].
    ^CHeapStrings at: CHeapStringsIndex.
```

Finally, the only method that needs to be published to WindowsInterface subclasses is:

```
copyToStaticHeap: aString

    | len cstr |
    len := aString size.
    cstr := self cHeapString.
    cstr copyAt: 0 from: aString size: len startingAt: 1.
    cstr at: len put: 0.
    ^cstr
```

This method accepts a Smalltalk string, copies it to the next available cHeapString, and returns the cHeapString pointer.

The following method illustrates the use of the cHeapString feature described above and extends our profile string examples by providing access to an application-specific .INI file.

```
getPrivateProfileString: sectionString
    entry: entryString
    default: defaultString
    fileName: nameString

    | returnString |

    returnString := self class cHeapString.

    self GetPrivateProfileString: (self copyToStaticHeap:
sectionString)
        with: (self copyToStaticHeap: entryString)
        with: (self copyToStaticHeap: defaultString)
        with: returnString
        with: (self class cHeapStringSize)
        with: (self copyToStaticHeap: nameString).
    ^returnString copyCStringFromHeap
```

Compare this code to getProfileString:entry:default described at the beginning of the article.

Is all this trouble worth it? Probably not if you are just making a few calls to determine configuration at startup. However, if your Windows interface is extensive and you make many calls to it, the answer is probably yes. The following tests were executed three times each.

```
Time
    millisecondsToRun:
        [100 timesRepeat: [WindowsInterface
copyToStaticHeap: 'Test'] ]
18 17 20

Time
    millisecondsToRun:
        [100 timesRepeat: ['Test' gcCopyToHeap16] ]
125 117 130
```

As you can see, the optimized method is six or seven times faster at runtime and does not require an incremental garbage collection performance penalty. The only garbage collection required by this optimization is at quitting time. These are truly cheap strings!

The catch here is that you can only make as many as eight calls to copyToStaticHeap: per Windows function invocation. Otherwise, your ninth parameter will overwrite the first. Practically speaking, this is not a problem. It does not appear that any of the Windows function calls contains more than eight string parameters. You could also make the queue larger than eight. And there are ways to make the queue dynamically expandable.

Dayle Woolston and Bob Capel have been working with Smalltalk for four years building client/server database applications. They can be reached at dayle_woolston@novell.com and bob_capel @novell.com.

Faster queries in Smalltalk

Jay Almarode

IN A PREVIOUS COLUMN, I described how to express various queries using the select:, detect:, and reject: methods. When the argument to these methods is a regular block (delimited by [and] brackets), the default implementation enumerates over every element of the collection, sending messages to determine if the element satisfies the query criteria. An alternative kind of block, called a SelectBlock (delimited by { and } brackets), restricts the kinds of statements inside the block, but also allows the use of a dot notation to specify the traversal along a path of instance variables. The main advantage of using SelectBlocks is that users can create indexes on certain kinds of collections, and SelectBlocks can take advantage of those indexes to speed up queries. This column describes indexes and how to use them effectively.

Before describing how to use indexes, what exactly is an index? An index provides a means to traverse backwards along a path of instance variables for every object in the collection for which the index was created. For example, if you have a set of employee objects, then creating an index along the path 'spouse.occupation.title' will create internal indexing structures that maintain mappings from titles (a String) to occupations, from occupations to spouses (a Person), and from spouses to employees in the set. With this information, when a query is posed to find all employees whose spouse's title is 'Manager', the indexing subsystem searches for all 'Manager's, then works backwards to obtain the employees. This is usually much quicker than iterating through the entire set of employees, traversing through all spouses and occupations to find the ones that match.

In SmalltalkDB, there are two kinds of indexes that users may create: identity and equality indexes. For either kind, when creating an index, you must specify an index path, which is the path of instance variable names for which the reverse mappings will be maintained. To create an index for the previous example, you might perform the statement:

```
aSetOfEmployees createEqualityIndexOn:
    'spouse.occupation.title'.
```

Once the index is created, queries that contain a predicate using this index path can use the reverse mappings to speed them up. The two kinds of indexes are used for slightly different purposes. Identity indexes are used to speed up queries where the comparison operator is based upon identity, i.e., when the comparison operator is == or ~~. These kinds of queries are unique to object-based systems where objects have unique identity. For example, if you have a reference to the instance of Department that represents the sales department (stored in the global variable SalesDept), then you could express the query to find all employees that work there by performing:

```
aSetOfEmployees select:
    { :emp | emp.worksIn == SalesDept }
```

The other kind of index that can speed up queries is an equality index. An equality index is used to speed up range queries and queries that utilize an equality operator. A range query is a query that contains a comparison operator like <, >, <=, or >=. For example, to find all employees over 30 years old and less than 50 years old, you could perform the query:

```
aSetOfEmployees select:
    { :emp | (emp.age > 30) & (emp.age < 50) }
```

In this example, both predicates traverse the same path, so an equality index on that path could be used if it exists. In fact, the indexing subsystem recognizes that the two predicates define an upper and lower bound on the values to be searched and combines the two lookup operations when composing the result. One thing to note about identity and equality indexes is that the existence of one kind of index does not prohibit you from expressing any kind of query. If an index is available for the query path and kind of comparison operator, then the indexing subsystem will utilize it; otherwise it will evaluate the query by enumeration. You can find out the available indexes on a collection by sending it the message equalityIndexedPaths or identityIndexedPaths, both of which return an array of strings.

Now that you know how indexes are used, let's look under the hood and see how they are implemented. To maintain the reverse mappings along an index path, the indexing subsystem must maintain auxiliary structures that provide very fast lookup operations and are built to handle large numbers of objects. These structures are typically hash dictionaries and btrees. In addition, the indexing subsystem must handle the updating of these structures when an object that participates in an index is modified or when new objects are made to participate in

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

an index. For identity indexes, all reverse mappings along the index path are maintained in a special kind of identity hash dictionary called an RcIndexDictionary. This dictionary is optimized for a very large number of entries, as well as having "reduced conflict" characteristics. In a previous column, I described reduced conflict classes and how they allow concurrent users to modify the same object without experiencing concurrency conflict. An RcIndexDictionary allows concurrent adders and removers to update the dictionary without experiencing conflict. Because this dictionary may be updated whenever a user adds or removes an object to the indexed collection, or modifies an instance variable of an object that participates in an index, it is important to reduce concurrency conflicts.

For equality indexes, all the reverse mappings except the last are maintained in the collection's index dictionary. The last reverse mapping in the index path is maintained in a btree. For example, if there were an equality index on a set of employees along the path 'spouse.address.zipCode' then the reverse mappings for spouse to employee and address to spouse would be maintained in the index dictionary, and the mapping from zipCode to address would be stored in a btree. Btrees are a standard indexing structure used in databases because they have many desirable characteristics. Btrees maintain an ordering of its entries in such a way that the tree is always balanced; all nodes in the tree have at least a minimum number of entries and all leaf nodes in the tree reside at the same depth. Because a node can hold many entries, btrees are usually shallow, so search operations are quite fast. When you execute the "selectAsStream:" query (described in my previous column), the stream object that is returned as the result of the query is actually streaming over the contents of the btree.

For more sophisticated users of indexes, a new feature allows developers to plug in their own custom btree nodes to speed up queries. In the latest release of GemStone, users are allowed to create equality indexes along a path where the last object is an instance of a user-defined class. Prior to this feature, the last object along the path had to be one of a few basic kinds of objects, like Strings, Numbers, Characters, etc. This was necessary to ensure that the last objects along the path could be ordered. To allow the last object to be a user-defined object, you are required to implement the comparison operators (<, >, <=, >=, and =) for that class. These operations must be implemented so that the following properties hold:

If $a < b$ and $b < c$, then $a < c$

Exactly one of these is true: $a < b$, or $b < a$, or $a = b$

$a <= b$ if $a < b$ or $a = b$

If $a = b$, then $b = a$

If $a < b$, then $b > a$

If $a >= b$, then $b <= a$

Adhering to the properties described above is the only requirement to be able to create equality indexes on paths where the last object is a kind of user-defined class. However, you can go one step further by emulating the technique that is used for storing basic kinds of objects (Strings, Numbers, etc.) in btree nodes. This technique involves storing additional information in a btree entry so that the lookup operations execute faster. For example, when GemStone stores a string in a btree, it also stores an encrypted form of the first nine characters of the string. When performing look-up operations, if the first nine characters determine whether the entry is <, >, or = to the lookup key, then it is not necessary to fetch the string to perform the comparison operation. Obviously, the fewer objects that you have to fetch to perform comparisons, the faster the total search operation will execute.

So how do you create your own custom btree nodes that store additional information in an entry? It is mainly a matter of creating subclasses of existing btree node classes, and overriding a small number of methods. I will illustrate with an example class called BagOrderedBySize, a subclass of Bag whose ordering is determined by its size (i.e., a bag with the two elments is less than a bag with five elements). As you might expect, BagOrderedBySize implements the comparison operators based upon the sizes of the receiver and the argument. To speed up comparisons in the btree nodes, we will store the size of the bag in the btree entry. Thus, a btree entry will consist of the BagOrderedBySize, the object to which it is mapped (the reverse mapping), and the size of the bag.

The first thing to do is to create subclasses of BtreeInteriorNode and BtreeLeafNode, calling them CustomBagInteriorNode and CustomBagLeafNode. We need to override about 10 methods to make our custom btree nodes work; however, most of the methods are trivial. For example, we must override methods that return the btree entry size (it should return 3), and the encryption size (it should return 1, in which to hold the bag's size). Another method to override is the method that returns the maximum number of entries in a node. For performance reasons, we do not want the size of a btree node to be larger than a page, which is approximately 2,000 object references. Therefore, we override this method to return 667 (size of an entry times 667 = 2,001). We also need to override the method that returns the minimum number of entries a node should hold; for btrees this is typically half the maximum number of entries, so our method will return 333. Another trivial method we need to override is the one that returns the class of btree node to use as a parent node. This is used when we must split a node into two nodes and create a new node as a parent. In our case, this method returns the class CustomBagInteriorNode.

The remainder of the methods to override for our custom btree nodes involve inserting the size of a bag into a btree entry and performing comparison operations. The method to insert the size of a bag into a btree entry is as follows:

```
method: CustomBagLeafNode
_insertEncryptionFor: aBagOrderedBySize
value: aValue startingAt: offset
" Get the size of <aBagOrderedBySize> and place it in the
```

receiver at the given <offset>. The argument <aValue> is unused. "
self _at: offset put: aBagOrderedBySize size

There are four comparison methods that need to be overridden for our custom btree nodes. The implementation of one of them should illustrate how to implement the others:

```
method: CustomBagLeafNode
    _compareKey: aBagOrderedBySize lessThanEntryAt: offset
    " Perform a 'less than' comparison between <aBag-
OrderedBySize> and the entry whose key is at the given
<offset>. The size of the key has been stored at offset + 1. "
    " define comparison such that nil is less than any
BagOrderedBySize "
aBagOrderedBySize isNil
    ifTrue: [ ^ (self _at: offset) notNil ].
^ aBagOrderedBySize size < (self _at: offset + 1)
```

There are a few more details to integrate our custom btree nodes into the indexing framework. We must override a method on BagOrderedBySize to indicate which class of btree node to use when an index is created. This class method, called btreeLeafNodeClass, returns CustomBagLeafNode. Finally, because our comparison operations are based upon the size of the bag, when an object is added or removed from any particular bag, the ordering of bags in our btree nodes might change. Therefore, we must update any btrees in which the bag is stored when a change to the size of the bag occurs. Fortunately, there are convenience methods that do this work for us. There are two methods, inherited from Object, one of which removes the receiver from any btrees in which it is stored, and another that inserts it back. The implementation of the add: method for BagOrdered-BySize illustrates the use of these methods.

```
method: BagOrderedBySize
add: newObject
" Adds <newObject> to the receiver, updating any btrees
in which the receiver might stored."
| values |
values := self removeObjectFromBtrees.
super add: newObject.
self addObjectToBtreesWithValues: values
```

Hopefully this column has given you a deeper understanding of how indexes work. They are most useful when querying large collections where iteration over the entire contents is prohibitive. For small collections, the overhead of determining which index to use and fetching the internal indexing objects negates the efficiency of indexes. For collections that contain more than 1,000 objects, indexes can speed up your queries. Another factor in determining if indexes will help is the percentage of objects returned for a query. If all elements of the collection satisfy the query, then you have effectively touched all objects the same as if you had enumerated over the entire collection. In this case, indexes will not give you any performance advantage over brute force enumeration. To summarize, indexes give the best performance for large collections and for queries with high selectivity.

MANAGING ASYNCHRONOUS METHODS *continued from page 11*

The source code for the ExternalReadStreamAdaptor is within the Manchester Goodies Library and its mirror sites, registered in the file: ExternalReadStream_st within the VisualWorks directories. The reader can also obtain the source directly from the author. We hope this class will be of use to other developers and welcome any comments.

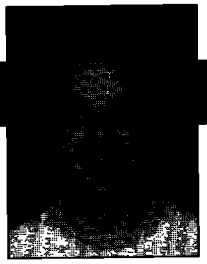
Acknowlegment

The author wishes to thank Greg Cowin and Shawn Smith for their feedback and suggestions in the design of the services described in this article.

References

1. Teale. C++ IOStreams Handbook, Addison-Wesley, Reading, MA, 1993.
2. ParcPlace Systems Inc. Advanced Programming Users' Guide, Ch. 3, p. 9.
3. White, Deugo, and Ulvr. Object transfer between Smalltalk VMs, The Smalltalk Report 4(2):11–13, 1994.
4. ParcPlace Systems Inc. ObjectWorks \ Smalltalk Users Guide, Ch. 8, p. 81.

Dr Michael Christiansen is a Consultant for Bell-Northern Research in Richardson, TX, where he develops management applications for cellular telecommunication networks. He can be contacted through BNR at 214.684.2550 or by email at mikec@metronet.com.



Kent Beck

Uses of Variables: Temps

THE TOPIC OF this and the next column is how to use instance and temporary variables. I came upon the information (some of it is patterns, some isn't yet) in the course of writing my forthcoming book of Smalltalk coding patterns. I looked at temporary variables first, and found patterns for four ways temps are used:

1. Reuse the value of a side-effecting expression (like ^SStream>>next^T)
2. Cache the value of an expensive expression
3. Explain the meaning of a complex expression
4. Collect the results of several expressions

Success in hand, I went to look for a similar set of canonical uses of instance variables. No dice. I came up with a taxonomy of the uses of instance variables, but no patterns. Many of the uses are bound up in other patterns (like the instance variable that holds the current State object). Others are too vague to make good patterns. I'll present what I have so far, because I have found it useful even in its unpolished state.

Ward Cunningham and I had a good long talk about this. We decided that the reason I was having so much trouble was scope. In the first book I am looking for coding patterns, the tactics of successful Smalltalk. Choosing to use a temporary variable is a tactical decision. It affects nothing but the single method in which the variable is used. Choosing to use an instance variable is not a tactical decision (except in a few cases like caching). Instance variables are tied up with the bigger issues of making models in Smalltalk. I already had lots of patterns upstream and downstream of the temporary variable patterns, so they fit right in. The modeling patterns are not nearly so well developed (that's why I'm leaving them out of the first book).

PPDUC

Before I talk about variables, I'd like to give you an update from PPDUC.

I gave a half-day pattern tutorial the first day. Around 150 folks attended. I knew I wanted to cover a lot of ground quickly, so I tried something new for me: I programmed live while talking about coding patterns.

Let me recommend this as a technique to all you trainers out there. The great thing about programming and teaching at the same time is there is so much shared context. You create a class, then ask, "How are we going to represent instance creation?" Everybody is thinking about what they'd do, so when you introduce the pattern (Complete Creation Method), they can see exactly how it relates to their experience. The terrible thing about programming and teaching simultaneously is that you are trying to keep two stories going in your head at once—the development story and the teaching story. I found myself typing a few characters, talking for 15 seconds, typing a few more characters, talking some more. Really quite distracting.

I had been hacking like crazy trying to get a new release of Profile/V ready for the show (I didn't quite make it), so I was running on little sleep. The morning I left for the conference my wife reminded me "don't say anything you don't want to see printed in THE SMALLTALK REPORT." I'm afraid I blew that in the first couple of minutes. Oh well...

On a related note, a very angry developer came up to me at one of the breaks. A colleague of his had read some of my comments from Smalltalk Solutions in THE SMALLTALK REPORT, and understood them to mean that you don't have to design Smalltalk programs. The VAD blamed me for the resulting mess, which was now his problem since he had inherited the code.

I won't even argue about whose responsibility the ugly code is. If even one person misunderstood, however, the comment deserves a little explanation.

You have to design Smalltalk programs much *more* than programs in other languages, not less. You expect Smalltalk programs to *do* much more. However, you can't do all that design at the beginning of the project when you're ignorant. You have to get smart before you design. Effective design happens in *episodes* (Ward's word) throughout the life of a project. Just because you don't kill a whole bunch of trees in the first six months doesn't mean you aren't designing.

Back to the conference, many of the presentations were deadly dull. Next year I expect much better. I see lots of amazing things out there, so I know there's enough material. Two stand-outs were Ward's talk about how to decide to harvest frameworks from code and Roby's 1200cc presentation of "Smalltalk: The Web Server." Coming soon to a network protocol stack near you!

The booths were certainly lively. Everybody had lots of traffic. The attendees seem to be serious about looking for ways to protect and capitalize on their Smalltalk investment. That's good news to us third-party folks.

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).

TEMPORARY VARIABLES

Smalltalk provides temporary variables to hold objects for the duration of a method. When I learned Smalltalk, I had to learn how to use them by reading examples in the image. Trial, error, and reading left me with a handful of ways to use temps, but the information was all subconscious. When I was looking for patterns of use, I went through every method in the image that uses temps:

```
| methods |
methods := OrderedCollection new.
Object allSubclassesDo:
 [:eachClass |
    eachClass selectors do:
       [:eachSelector | | node |
       node := eachClass decompile: eachSelector.
       node block body temporaries notEmpty ifTrue:
[methods add:
eachClass printString , ' ' , eachSelector]]].
MethodListBrowser
    openListBrowserOn: methods
    label: 'Methods with temps'
```

I cycled through all the methods trying to classify each temporary variable. When I got to a variable I couldn't classify, I added a new category. Here are the results, written as patterns:

Pattern: Reusing Temporary Variable

How do you repeatedly use the same evaluation of an expression whose value changes?

Repeating an expression is often the simplest way to write a method. Reading a method without temporary variables is easier than reading one that has them. Occupying your mind remembering the assumed value of a variable provides a distraction from the work of comprehending the rest of the method.

Some expressions return new values for each evaluation because of side-effects. If you are relying on the same value, you cannot simply execute the expression again and get the right results. For example:

```
parseLine: aStream
    aStream nextWord = ^Qone' ifTrue: [^self parseOne: aStream].
    aStream nextWord = ^Qtwo' ifTrue: [^self parseTwo: aStream].
    ...
```

is not likely to be what you meant. Instead, you want to grab the first word in the line once, then use that word in subsequent tests:

```
parseLine: aStream
    | keyword |
    keyword := aStream nextWord.
    keyword = ^Qone' ifTrue: [^self parseOne: aStream].
    keyword = ^Qtwo' ifTrue: [^self parseTwo: aStream].
    ...
```

Other expressions, like "Time millisecondClockValue", change value because of resources external to Smalltalk rather than

side-effects. They, too, must be stored in a Reusing Temporary Variable.

Store the value of the expression in a temporary variable. Use the variable instead of the expression in the remainder of the method.

Pattern: Explaining Temporary Variable

How do you clearly communicate the intent of complex expressions?

Introducing the complexity of a temporary variable may be worth the cost if there are complex expressions in a method. Readers must carefully study an expression with 5 or 10 messages embedded in it to understand its meaning. You can use a temporary variable to communicate the intent of part of the expression.

For example, you might need to compute the size of a widget by combining several factors:

```
extent
    ^self textWidth + self leftBorder + self rightBorder +
self margin
@ (self textHeight + self topBorder + self bottomBorder +
self margin)
```

Compare that to:

```
extent
    | x y |
    x := self textWidth + self leftBorder + self rightBorder + self
margin.
    y := self textHeight + self topBorder + self
bottomBorder + self
margin.
    ^x @ y
```

You can read the second version in three separate chunks, without having to understand the whole expression at once.

Store the value of a part of a complex expression in a temporary variable. Use the variable in place of the subexpression. Give it a name that reflects the meaning of the expression.

Explaining Temporary Variables are often a prelude to Composed Method. The example above looks even better as:

```
extent
    ^self width @ self height
```

Pattern: Caching Temporary Variable

How do you improve the performance of a method that repeatedly calculates the same value for an expression?

Often, redundant calculation makes for the most readable code. For example, if you haven't had any other excuse to introduce a temporary variable, you shouldn't use one just because you are redundantly executing an expression. For example, self bounds in the following code always returns the same Rectangle, but it reads best if it is executed repeatedly:

```
smallerChildren
    ^self children select: [:each | self bounds contains:
each bounds]
```

If you measure that the repeated execution of bounds is slowing the whole computation down, and if this method is the only one for whom it is a bottleneck, the simplest solution is to compute it once and store the value in a temporary variable:

```
smallerChildren
    | bounds |
    bounds := self bounds.
    ^self children select: [:each | bounds contains: each
        bounds]
```

The result is longer, more complex (because of the temp you have to keep track of), and more prone to breaking (what happens if the receiver's bounds change during the method?) However, if you need the code to go faster, the costs are likely to be a good investment.

Execute the expression once. Put its value in a temporary variable. Use the variable instead of the expression.

Pattern: Collecting Variable

How do you collect results across several expressions?

The enumeration protocol does a good job of relieving you of the burden of writing most looping code. You just write collect or select or whatever and the details are taken care of for you.

This is all well and good as long as you are working with a single collection at a time (which is 95% of all uses). When you need to coordinate several collections, or even collect results from several objects, you need to do a bit more of the coding yourself.

As with the other temporary variable patterns, if you can get away without them, you should. However, the only alternative in this case is to write a whole slew of enumeration methods, and keep extending them for every new application. Using a temp isn't so bad compared to that.

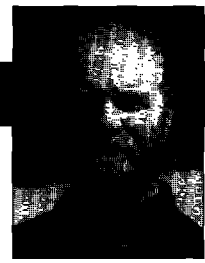
For example, say you need to return the concatenation of two collections, but the elements should be perfectly shuffled—an element from the first, an element from the second, an element from the first, and so on. Here's how you do it using a temporary variable:

```
couples
    | result |
    result := OrderedCollection new.
    self girls
        with: self boys
        do:
            [:eachGirl :eachBoy |
            result add: eachGirl.
            result add: eachBoy].
    ^result
```

Use a temporary variable to collect results. Initialize it, add to it, and return it as the value of the method.

CONCLUSION

In the next column I'll talk about the 11 ways I've found so far for instance variables to be used. See you next month!




Jan Steinman Barbara Yates

Managing project documents

In the June issue, we made a case for "continuous documentation," and outlined what that entails. We also promised to give you some concrete examples and source code, so you could begin to implement a continuous documentation process. In the July issue, as many readers know, we were diverted from this topic by a base image change alert. So, now let's return to the topic of managing project documents.

There are at least five widely differing Smalltalk dialects out there, augmented by two major and numerous minor code management systems. Rather than attempt the impossible task of embracing such diversity, we're presenting stuff that is actually implemented and working in VisualWorks 2.0 under ENVY/Developer R1.43.

Many of these things can be done in other environments. However, much of the following assumes you can associate storage with arbitrary software components, which might be difficult if your code manager is simply a layer on top of source code files.

- **Principle 1: Conceptual Integrity**—Documentation must be at the same level as that which it describes. There is simply no way you can cram all your documentation needs into class and method specifications. ENVY provides nestable modules called *Applications* and *SubApplications* that have a specification field, and a module binding component called a "configuration map" that also has a specification field.
- **Principle 2: Constant Accuracy**—Documentation must be stored with that which it describes. If you want your developers to maintain their documents, you've got to make it easy for them to do so.
- **Principle 3: Accessibility**—Documentation must be available quickly and efficiently from other, related documentation. This does not mean sending a reference number via email to your organization's technical library!

A SIMPLE MACRO FACILITY

The marriage of these three principles demands some way of linking related parts of documentation together. VisualWorks has a simple but efficient tagged-character class, Text, that allows you to associate arbitrary objects with each character in a string. (If your Smalltalk has no such thing, you will need to either add it, or come up with some other linking mechanism.)

This tagged-character capability suggests a simple "not-quite-hypertext" linking facility. First off, we need to fix a bug; add the following instance method to TextStream:

```
TextStream:
nextPutAll: aText
    "Place each of the elements of aText on myself, starting
     at my current position. If I'm fed an instance of Text,
     keep its emphasis. Answer aText."

    (aText respondsTo: #runs)
        ifFalse: [^super nextPutAll: aText].

    1 to: aText size do: [:i |
        "I know, it's brute-force. Someday, this should be
         optimized by examining runs."
        self
            emphasis: (aText emphasisAt: i);
            nextPut: (aText string at: i)].
    ^aText
```

Without this bug fix, Text fed to a TextStream loses its emphasis, which sorta defeats the purpose! Readers of our last column may recognize a "signature testing" pattern of managing system changes, and may recall that an override *is* indeed a base image change. Be aware that code that expects the superclass behavior will now be "broken" by this bugfix!

Now implement the following three methods in a class extension of Text. We try to put system additions in a parallel application with a similar name. In our repository, the following extensions are in a subapp called DevelopmentBytesmiths because they relate to the development process, and not particularly to Text in its full generality. Note also that these extensions are dependent on Compiler; simply adding these extensions to the subapp Collections, where Text is defined, would create a circular dependency between the apps Kernel and Compilation.

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

```
Text:
withInclusions
    "Answer a copy of me in which all strings with the
     emphasis #Smalltalk are evaluated and replaced with
     a String or Text representation of the result."

    ^self withInclusionsIn: nil

withInclusionsIn: codeOwner
    "Answer a copy of me in which all strings with the
     emphasis #Smalltalk are evaluated and replaced with
     a String or Text representation of the result. The
     context of evaluation is the object codeOwner."

    ^(self
        withInclusionsIn: codeOwner
        on: (TextStream on: (String new: self size * 2 "Guess
            that inclusions may double size."))) contents

withInclusionsIn: codeOwner on: textStream
    "Place on textStream a copy of me in which all strings
     with the emphasis #Smalltalk are evaluated and
     replaced with a String or Text representation of the
     result. The context of evaluation is the object
     codeOwner. Answer textStream."

    | whereAmI whereWasI |
    ^(self size > 0 and: [runs values includes: #Smalltalk])
        ifFalse: [textStream nextPutAll: self]
        ifTrue:
            [whereAmI := whereWasI := 1.
            [whereAmI := whereAmI + (self runLengthFor:
                whereAmI).
            (self emphasisAt: whereWasI) == #Smalltalk
                ifFalse: [textStream nextPutAll: (self copyFrom:
                    whereWasI to: whereAmI-1)]
                ifTrue: [(Object errorSignal
                    handle: [:ex |
                        textStream
                            emphasis: #bold;
                            nextPutAll: '*** Cannot include the
                                following expression!! ***';
                            emphasis: nil;
                            cr;
                            nextPutAll: (string copyFrom: whereWasI
                                to: whereAmI-1).
                        ex returnWith: '']
                    do:  [Compiler
                            silentEvaluate: (self copyFrom:
                                whereWasI to: whereAmI-1)
                            for: codeOwner
                            logged: false]) withInclusionsIn:
                                codeOwner on: textStream].
            whereWasI := whereAmI.
            whereAmI = 0 or: [whereAmI > self size]]
                whileFalse: [].
            textStream]
```

The VisualWorks 2.0 compiler is in many ways as old as Smalltalk itself, and has not been fully integrated with signals and exceptions. We fixed that by adding **silentEvaluate:for:logged:**, which always raises an exception when compilation breaks. You can do the same, or you can change this to **evaluate:for:logged:** and put up with the occasional syntax error window when expanding Text that has bad expressions to expand.

This facility is recursive; if a section of Text has the emphasis #Smalltalk, the resulting expansion is itself scanned for inclusions, so any object that understands **withInclusionsIn:on:** can implement its own expansion scheme. In fact, without someone putting an end to this recursion, there can be real trouble! Implement the following two methods in extensions to Object and String, respectively:

```
Object:
withInclusionsIn: ignored on: textStream
    "Place on textStream a printable representation of me
     suitable for use in documentation. Answer textStream.
        This is a 'bug catch' message in this class, and should
        normally only be sent to Texts or Strings. Subclasses should
        not normally override simply to change presentation."

    self printOn: textStream.
    ^textStream
String:
withInclusionsIn: ignored on: textStream
    "Place myself on textStream for use in documentation.
     Answer textStream."

    textStream nextPutAll: self.
    ^textStream
```

For even more flexibility, add the following to an extension of BlockClosure. If the Text being expanded is a block, the block will be evaluated, and the result will be inserted into the expanded output. The block can optionally take the "code owner" and the active TextStream as arguments, which allows included source code blocks to query their environment and directly manipulate the resulting expanded output.

```
BlockClosure:
withInclusionsIn: codeOwner on: textStream
    "Place on textStream a printable representation of my
     evaluation suitable for use in documentation. Depending
     on the number of arguments I take, pass me codeOwner
     and textStream on evaluation. Answer textStream."

    | args |
    args := Array new: self numArgs.
    1 <= args size ifTrue: [args at: 1 put: codeOwner].
    2 <= args size ifTrue: [args at: 2 put: textStream].
    ^(self valueWithArguments: args) withInclusionsIn:
codeOwner on: textStream
```

These methods add the basic "hypertext" behavior to Text. Now your app/subapp comments can have things like

"MyClass comment" with the emphasis #Smalltalk, and sending **withInclusions** to that Text will embed the comment for MyClass. But, by itself, this inclusion facility is not terribly useful for two reasons:

1. ENVY strips the per-character attributes from Text before storing it.
2. Basic VisualWorks provides no user interface for applying custom character attributes to a Text.

ENVY Text storage

If your hypertext goes away when your image quits, it won't improve your team's productivity one bit! When you press the "source" button in an ENVY browser to switch to "comment" mode, any changes you make are stored as Strings in "inherited user fields." These are arbitrary key-value storage locations, in which both key and value must be a String. If we can trick ENVY into storing Text (or even arbitrary objects) in these fields, it saves us from changing each of the many places where these fields are accessed.

This gets a little difficult, because the source code for methods that access the repository has been removed, and your ENVY/Developer license keeps you from decompiling or otherwise reverse-engineering those methods. The following technique allows you to copy the hidden methods and associate that copy with a new method selector, allowing you to provide an original implementation in its place that conditionally sends the old method.

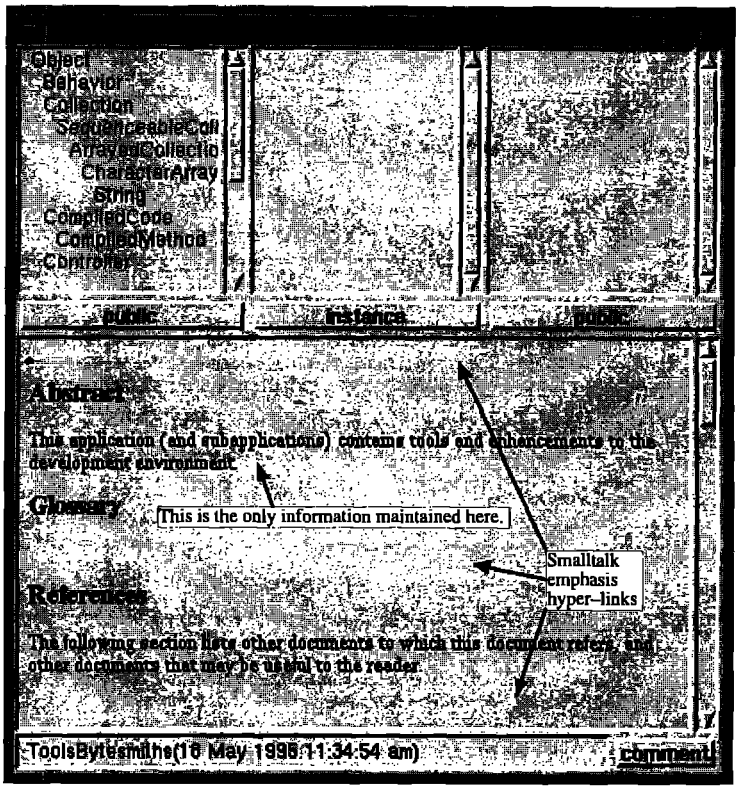


Figure 1. An application comment template, showing hyper-links.

OTI has no *legal* objections to this technique, but it can be dangerous if misused! We've been using the following for some time, but if there is a typo, or if you use this technique to intercept and modify other hidden methods, you may damage your ENVY repository. Be sure to follow our suggestions in last month's column for managing base image changes, and consider making and testing these changes in a separate repository until you are certain they are safe.

Do this in a workspace to copy and rename the two hidden methods that need to be intercepted:

```
| meth oldRecord |
meth := (UserFieldRecord compiledMethodAt: #contents)
    copy.
oldRecord := meth record.
meth selector: #contentsFromVendor.
UserFieldRecord
    updateEditionsRecordIn: LibraryManagement
    with: [:record | record
        addMethod: meth
        source: nil
        basedOn: oldRecord
        changeCategoryTo: 'intercepted methods']
    ifUnable: [].
meth := (Record class compiledMethodAt: #libraryFormatFor:)
    copy.
oldRecord := meth record.
meth selector: #libraryFormatForFromVendor:.
Record class
    updateEditionsRecordIn: LibraryManagement
    with: [:record | record
        addMethod: meth
        source: nil
        basedOn: oldRecord
        changeCategoryTo: 'intercepted methods']
    ifUnable: []
```

Before we go any further, we need to establish the predicate we are going to use for switching between the original implementation and our Text-capable implementation. Put the following two methods in the same Application or SubApplication where you put the other class extensions:

```
Text class:
canReadFrom: chars
    "Does the String (or streamed String) chars contain
     information that can be used to create an instance of
     me via #readFrom:?"

    | signatureChars |
    ^chars size >= "String new asText storeString size" 56 and:
        ['(Text string: '''
            occursIn: (chars isSequenceable
                ifTrue: [chars]
                ifFalse:
                    [signatureChars := chars next: "(Text
                    string: '' size" 15.
                    chars position: chars position - 15.
                    signatureChars])
                        at: 1]
```

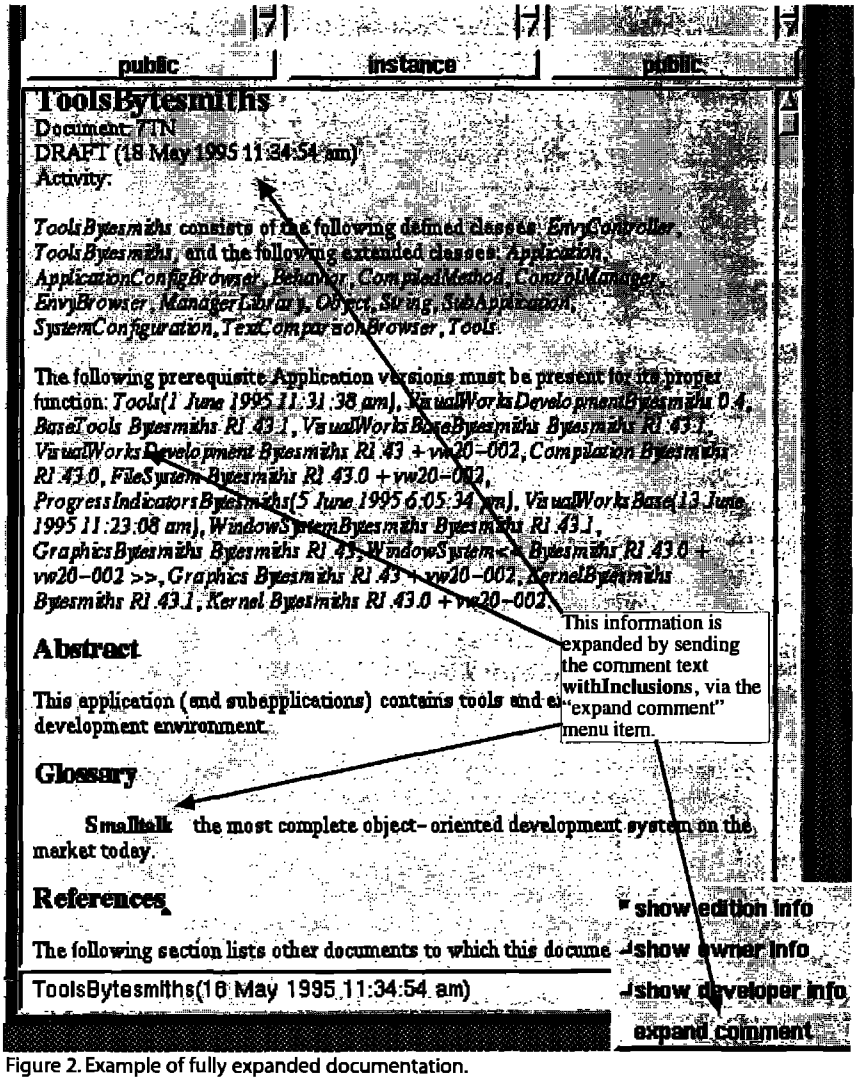


Figure 2. Example of fully expanded documentation.

Text
libraryFormat
"Answer a representation of myself suitable for storing in ENVY user fields."

self storeString

Now replace those two hidden methods that we copied with original implementations that conditionally send the renamed method:

UserFieldRecord:
contents
"Answer my contents, decoding them if necessary."

| charStream decoder |
charStream := self collection readStream position: self startPosition - 1.
^((Text respondsTo: #canReadFrom:) and:
[Text canReadFrom: charStream])
ifTrue: [Text readFrom: charStream]
ifFalse: [self contentsFromVendor]

Record class:
libraryFormatFor: *anObject*
"Answer a format suitable for storing *anObject* in the library."

^(anObject respondsTo: #libraryFormat)
ifFalse: [self libraryFormatFor
FromVendor: anObject]
ifTrue: [anObject libraryFormat]

Note the conditional nature of these changes. They can safely replace those in the base image, because they will not break if the Application or SubApplication containing the Text extensions is not present. Now, character emphases you change in any comment or notes field will be preserved in ENVY. More importantly, the emphasized commentary does not "break" if these changes aren't present; since they are in the **storeString** format, they are simply a bit difficult to read.

You now have all the "modeling" changes you need to implement hypertext based on arbitrary Smalltalk expressions embedded in Text objects—that's always the hardest thing to get right. Now all you need is a UI!

Setting #Smalltalk Text emphasis

We've added extensive support for viewing, searching, and modifying these embedded expressions that we don't have room to show here. However, there are a few simple things you can do to get started, which may be enough to get you started.

ParagraphEditor is the class that generates and modifies Text, including character emphasis. Unfortunately, there is no simple way to fully support new emphasis types without changing existing ParagraphEditor code, making new TextAttributes and CharacterAttributes instances, and then managing those instances as long-lived system resources.

We're going to cheat by adding a simple facility for setting and removing the new #Smalltalk emphasis we've specified in a less general way. To be able to set or remove an arbitrary emphasis, add the following to an extension of ParagraphEditor:

ParagraphEditor:
addEmphasis: *emphasis*
"Add the given *emphasis* to the emphasis of the current selection."
| thisText |
thisText := self selectionStartIndex = self selectionStopIndex
ifTrue: [Text string: 'x' emphasis: emphasisHere]
ifFalse: [self selection].
thisText addEmphasis: emphasis

JUST PUBLISHED!

Rapid Software Development with Smalltalk

The Ultimate Guide to Better Smalltalk Development...Write Code Faster Without Sacrificing Quality.

RAPID SOFTWARE DEVELOPMENT WITH SMALLTALK covers the spectrum of O-O analysis, design, and implementation techniques and provides a proven process for architecting large software systems. By using detailed examples of an extended Responsibility-Driven Design (RDD) methodology and Smalltalk, readers will find techniques derived from real O-O projects that are directly applicable to on-going projects of any size.

The author provides readers with specific guidelines that could dramatically cut costs and keep projects on-time. Specifically, the author provides readers with project patterns that work, illustrations of design patterns, O-O metrics with example code to test design quality and of course, numerous Smalltalk code examples.

About the Author...

Mark Lorenz is the founder and president of Hatteras Software, Inc. a company that specializes in helping projects use object technology successfully. The author has already published two popular books on object technology entitled OBJECT-ORIENTED SOFTWARE DEVELOPMENT: A PRACTICAL GUIDE and OBJECT-ORIENTED SOFTWARE METRICS (Prentice Hall) and also writes a regular column for THE SMALLTALK REPORT called "Project Practicalities."

Readers will...

- Speed up the development process by fostering reuse
- Significantly reduce debugging time
- Gain step-by-step instruction on how to make the object model more robust
- Learn how to distribute responsibilities within the object model more effectively
- Discover a practical day-by-day breakdown of a rapid modeling session
- See how to organize the development team most efficiently

This book will prove invaluable to anyone interested in speeding up the consistent development of high-quality object-oriented software systems based in Smalltalk.

PART OF THE ADVANCES IN OBJECT TECHNOLOGY SERIES

Available at selected book stores. Distributed by Prentice Hall.

SIGS BOOKS ORDER FORM

❑ **YES!** Please rush me ___copy(ies) of RAPID SOFTWARE DEVELOPMENT WITH SMALLTALK at the low price $24 per copy. (ISBN: 1-884842-12-7; Approx 200 pgs.)

Money-back Guarantee: If I am not completely satisfied, I may return the book(s) within 14 days and receive a complete refund, promptly and without question.

❑ Check payable to SIGS Books
❑ Visa ❑ American Express ❑ MasterCard

Card#_____ Exp._____

Signature_____

SHIPPING AND HANDLING: For US orders, please add $5 for shipping and handling; Canada and Mexico add $10; Outside North America add $15. IMPORTANT: NY State residents add applicable sales tax. Please allow 4–6 weeks for delivery.

Name_____

Company_____

Title_____

Address_____

City/State/Zip_____

Country/Postal Code_____

Phone_____ Fax_____

SEND TO:
SIGS Books, P.O. Box 99425
Collingswood, NJ 08108-9970
Phone: 609.488.9602 Fax: 609.488.6188

SIGS BOOKS

HOW TO CONTACT SIGS PUBLICATIONS

To submit materials for publication
Article proposals, outlines, and manuscripts; industry news; press briefings; product announcements; letters to the editor—send to
John Pugh & Paul White, Editors
THE SMALLTALK REPORT
885 Meadowlands Drive, Suite 509
Ottawa, Ontario K2C 3N2, Canada
Phone: 613.225.8812 Fax: 613.225.5943
email: john@objectpeople.on.ca
paul@objectpeople.on.ca

For customer service and to order a subscription, renew, or change the name/address of an existing subscription
In the US—
P.O. Box 5050
Brentwood, TN 37024-5050
Phone: 800.361.1279 Fax: 615.370.4845
email: subscriptions@sigs.com

In Europe—
Subscriptions Dept.
Tower House
Sovereign Park
Market Harborough
Leicestershire, LE16 9EF, UK
Phone: +44(0)1858 435 302
Fax: +44(0)1858 434 958

To order back issues
Back Issue Order Dept.
SIGS Publications
71 West 23rd Street, 3rd floor
New York, NY 10010
Phone: 212.242.7447 Fax: 212.242.7574

For information on list rentals, contact:
Rubin Response
1111 Plaza Dr.
Schaumburg, IL 60173
Phone: 708.619.9800 Fax: 708.619.0149

For information on reprints of published material, contact:
Reprint Management Services
505 East Airport Road, Box 5363
Lancaster, PA 17601
Phone: 717.560.2001 Fax: 717.560.2063

To place an advertisement or request a media kit for any SIGS publications, contact:
Advertising Department
SIGS Publications
Phone: 212.242.7447 Fax: 212.242.7574

For information on SIGS Books and SIGS Conferences
Phone: 212.242.7447 Fax: 212.242.7574

```
        removeEmphasis: #()
        allowDuplicates: false.
    self selectionStartIndex = self selectionStopIndex
        ifTrue: [emphasisHere := thisText emphasisAt: 1]
        ifFalse: [self replaceSelectionWith: thisText].
    view topComponent refresh

removeEmphasis: emphasis
    "Remove the given emphasis from the emphasis of the
     current selection."

    | thisText |
    thisText := self selectionStartIndex = self
            selectionStopIndex
        ifTrue: [Text string: 'x' emphasis: emphasisHere]
        ifFalse: [self selection].
    thisText addEmphasis: #()
        removeEmphasis: emphasis
        allowDuplicates: false.
    self selectionStartIndex = self selectionStopIndex
        ifTrue: [emphasisHere := thisText emphasisAt: 1]
        ifFalse: [self replaceSelectionWith: thisText].
    view topComponent refresh
```

Now, wire it to a pair of "hot" keys. This method works like the other emphasis hot keys, such as "ESC b" to add bold and "ESC B" to remove bold, but unlike the method that implements other emphasis changes, this one is not hard-wired to a particular key.

```
ParagraphEditor:
changeInclusionKey: aChar
    "Add or remove 'inclusion' emphasis of the current
     selection, depending on the case of aChar. Uppercase
     removes, lowercase adds."

    aChar keyValue isUppercase
        ifTrue: [self removeEmphasis: #(Smalltalk)]
        ifFalse: [self addEmphasis: #(Smalltalk)].
    ^true
```

Finally, evaluate the following statements to bind this emphasis change to a "hot" key of your choice. (We chose "h" for "hyper," and because it was not already used.) These statements should go in the **loaded** method of the Application or SubApplication where you have been putting all this code. (While you're at it, add a **removing** method that undoes these key bindings.)

```
(ParagraphEditor classPool at: #Keyboard)
    bindValue: #changeInclusionKey: to: ESC followedBy: $h;
    bindValue: #changeInclusionKey: to: ESC followedBy: $H.
ParagraphEditor withAllSubclasses do: [:peClass |
        peClass allInstancesDo: [:pe |
            pe flushKeyboardMap]]
```

Viewing the results

Now you are able to create and store runtime inclusions

Introducing the new object technology magazine for Europe...

Upcoming Features

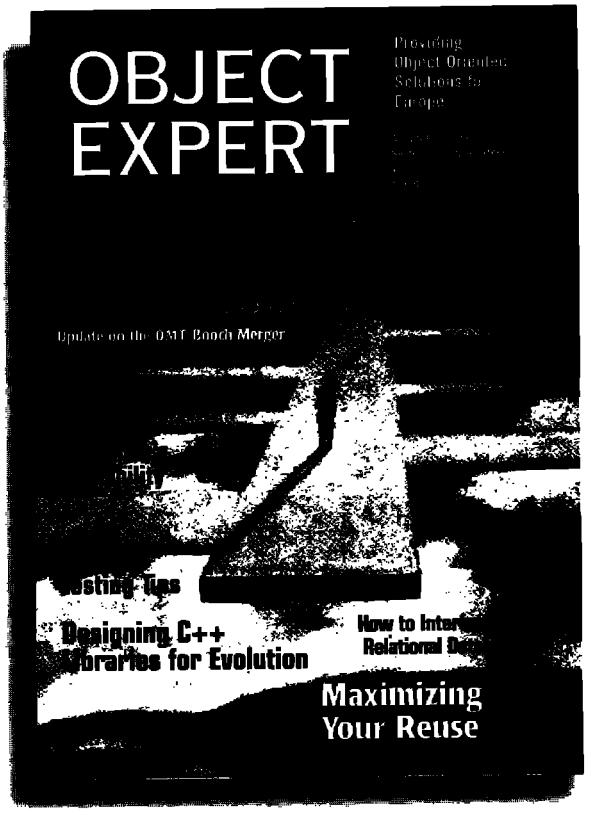
- What's Going On with Distributed Objects
- New Cost Estimation Techniques
- Mastering Memory Management
- Using Use Cases
- Business Objects Made Easy
- Building O-O Client/Server Architectures
- Understanding Object Database Systems

Regular Columns and Columnists

- Methods and Tools, ***John Cameron***
- Distributed Object Systems, ***Andrew Watson***
- Using C++, ***Bryan Boreham***
- System Architecture, ***Sanjiv Gossain***
- Smalltalk Means Business, ***Bernard Horan***
- Building From Components, ***Andy Carmichael***
- Migration Strategies, ***Alan O'Callaghan***
- Formal Techniques for OT, ***Alan Wills***
- Business Objects, ***Oliver Sims***
- Market Trends, ***Philip Carnelly***
- Skills Transfer, ***Roger Holmes***

A must-read for anyone who wants to become an Object Expert.

OBJECT EXPERT is written for and by European object technology users. Under the editorial direction of well-respected UK practitioner John Daniels, **OBJECT EXPERT** offers detailed how-to articles, insightful columns, and helpful case studies for those seeking object technology solutions. Each issue includes several short, easy-to-read articles for both newcomers and advanced users. **OBJECT EXPERT** is the one easily accessible and readable forum that unites all object technology users throughout Europe.



❑ YES! Send me one year of **OBJECT EXPERT** (six issues) beginning with the premiere November/December '95 issue:

❑ £25 UK ❑ £35 Europe ❑ £45 outside Europe ❑ DM 92 ❑ FF 324

METHOD OF PAYMENT

❑ Cheque enclosed *(Payable to SIGS, drawn on a bank in the country of currency)*

❑ Charge my: ❑ Visa ❑ Mastercard ❑ AMEX

Card#: _____ Exp. date: _____

Signature: _____

All credit card transactions will be debited at the £ Sterling rate above.
All prices include shipping.

Name _____

Job Title _____

Company _____

Address _____

Town _____

Country/Postal Code _____

Phone _____ Fax _____

FOR FASTER SERVICE,
call +44 (0)1858 435301
fax +44 (0)1858 434958

SIGS Publications, Tower House, Sovereign Park, Lathkill Street, Market Harborough, Leicestershire LE16 9EF United Kingdom

in your project documentation. This allows detailed documentation to be linked to abstract documentation, yet be maintained at the detailed level, while not distracting the reader of the abstract information. Figure 1 shows three hyperlinks in a code browser. We purposely make #Smalltalk emphasis difficult to read, to limit distraction and to encourage link expansion. But what if the reader *wants* to be so distracted? How do we see this stuff?

The simple way is to select a #Smalltalk-emphasized expression and inspect it. You get a basic inspector on a Text, which allows you to see the plain ASCII expansion. Alternatively implement an Inspector subclass for Text that allows you a WYSIWYG view of Text.

Rather than use up precious column space with an entire implementation, we'll tease you with some salient bits:

```
Inspector subclass: #TextInspector
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BaseToolsBytesmiths'

TextInspector comment: 'This class allows normal
ParagraphEditor style editing of a Text just by inspecting
it. It has no list view, since you don't really need one.'
TextInspector class:
view: anInspector in: area of: superView
    "Create a text view on anInspector in area of superView."
    superView
        add:
            (LookPreferences edgeDecorator on: (anInspector
                class textViewClass on: anInspector aspect: #text
            change: #acceptText:from:
            menu: #textMenu initialSelection: nil))
        in: area
```

The rest is eight methods that all override Inspector methods.

This works great as the "developer" interface—your developers will be so glad to be rid of keeping class comments in synch with app/subapp comments that they won't mind that their hyperlink to embedded documentation is an inspector. But something better is needed for the "linear reader" and for hard copy.

Unfortunately, any presentation change that is available from existing browsers is going to be a base image change. We considered wiring inclusion expansion to a new ParagraphEditor menu item, but we didn't feel the facility was general enough, and we certainly didn't want it slipping through to the end user!

Instead, we dug into the EnvyBrowser hierarchy, adding yet another mode to the code pane, with a menu item in the status area to expand all inclusions. Figure 2 shows the linearized hyperdocument, including two of the hyperlinks displayed in Figure 1. This isn't the entire story, but it should get you started:

```
AbstractMethodsBrowser:
expandComment
    "I assume that my text view is in comment mode.
     Cause the comment to be re-generated, with all Text
     of #Smalltalk style evaluated and inserted in-place."

    | oldState |
    [oldState := self textSelector.
    self changedTextTo:
#textShowingCommentWithInclusions]
        valueNowOrOnUnwindDo: [self textSelector: oldState]

textShowingCommentWithInclusions
    "Answer the expanded comment."

    ^Cursor read showWhile: [self commentString asText
        withInclusions]
```

CONCLUSION

Getting project documentation to flow out of the development process can greatly increase productivity, and if it is well done the developers actually begin to enjoy producing and maintaining their documentation!

In two columns, we've outlined the requirements and principles of a continuous documentation process, and provided some concrete examples of how it can be accomplished. In the future, we'll periodically revisit the topic with further design sketches and examples.