The Smalltalk Report

The International Newsletter for Smalltalk Programmers

October 1992 Volume 2 Number 2

Object Visibility: Making the Necessary Connections

By Rebecca Wirfs-Brock

Contents:

An exploratory design is by no means complete. It is a rough conceptual sketch of the key objects of a design, what their roles are, and a partial list of their responsibilities and collaborations. A lot of detail needs to be added before this relatively high-level design description can be turned into code. I want to focus on just one of those details: how to turn an imprecise list of collaborations into a more rigorous design description and finally into Smalltalk code—a relatively straightforward process that can be tackled systematically. Following a few general principles during this translation process results in classes that are more reusable and easier to change or enhance.

Turning an architectural drawing into a detailed set of blueprints shares a few similarities with the software construction process. When developing detailed blueprints, an architect translates a rough architectural drawing into a specific list of materials to be used and a fairly explicit map of how those materials should be composed in the finished product. This still leaves a lot of latitude for decision-making and creativity during construction—just ask anyone who has had a house built. You don't start construction expecting a barn and end up with a skyscraper! The same principles apply to constructing software.

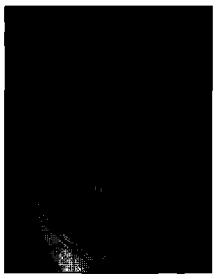
Before turning an object-oriented design description into a detailed set of software blueprints, you must consider the tools and environment you will use during construction. Mapping an object-oriented design into Smalltalk code requires matching up object-oriented design concepts with the *appropriate* Smalltalk language and programming constructs. It's essential when constructing a solution to have a good understanding of pre-existing components. Architects don't invent new kinds of fasteners or building material for each construction project. Similarly, proficient Smalltalk programmers know their Smalltalk class libraries. They don't construct a new class when a readily available one will do the job, even if it isn't perfect.

Before systematically adding more rigor to our collaborations, let's examine our Smalltalk construction environment. How many different ways are there in Smalltalk for one object to have visibility of another? Objects can't collaborate unless they can send each other messages. The client, or sender of a message, first needs to have visibility of the server or receiver of the message. Message sending is all done within the context of a method. Anyone with a modest amount of Smalltalk programming experience should be able to come up with most of these techniques fairly quickly. For new Smalltalkers this is a good exercise in fundamental implementation constructs. You will use these constructs (and other techniques) when you translate designs into executable program code.

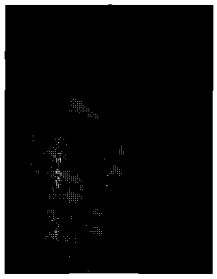
Here are some ways an object may be visible within a method:

- an object always has visibility of itself (sending messages to self is fundamental to Smalltalk programming)

continued on page 4



John Pugh



Paul White

EDITORS' CORNER

Since many of you will be reading this while attending OOPSLA'92 in Vancouver, we thought it appropriate to take stock of the impact OOPSLA has had on the growth of Smalltalk and vice versa. As we mentioned in our editorial last year following OOPSLA, we were struck by Smalltalk's "emergence" as an industrial-strength vehicle for large-scale object-oriented system development. Looking back, it's probably fair to say that was a new role for most of us. For the first time, it seemed we did not have to constantly defend the decision to use Smalltalk. For the first time, we regularly heard the question, Why *aren't* you using Smalltalk?

Both of the major Smalltalk vendors have major plans for OOPSLA. Digitalk has recently released the OS/2 version of PARTS Workbench, their long-awaited parts assembly and reuse tool set technology. Will this product lead us closer to the promised land of application construction from prefabricated software parts? ParcPlace will be showing VisualWorks, their new application development environment for client-server, GUI-based applications. With a growing number of third-party vendors also showing Smalltalk-related products, the OOPSLA exhibits floor will be an active place for Smalltalkers. Watch for reviews of many of these products in upcoming issues of the REPORT.

Once again, we feature Rebecca Wirfs-Brock's design column in the REPORT. This month, Rebecca describes the different ways one object can be visible to another and suggests guidelines for managing this visibility. In the long run, she suggests, it is vital for "teams to develop and stick to a style guide that addresses when and how to use parti- -ul Smalltalk constructs." Having faced these issues many times before on projects, we can only add that we agree wholeheartedly.

Also in this issue, Kent Beck introduces us to a number of collection idioms, illustrating how best to use Smalltalk's collection class library, which has traditionally been one of Smalltalk's best-selling features. Greg Hendley and Eric Smith return to their proposal for a three-layered architecture for building GUIs using a more complex example to highlight many of the pitfalls normally encountered during GUI development. Alan Knight rolls up his sleeves in this month's Best of comp.lang.smalltalk column and covers a number of very specific and technical questions relating to the implementation of Smalltalk. As he points out, many of the discussions he covers this month offer "only an understanding of the source of the problems" rather than solutions. Finally, Jan Steinman and Barbara Yates review ENVY Developer by Object Technology International. In our ongoing coverage of team programming tools, Jan and Barbara describe ENVY's philosophy and put ENVY's features into perspective with respect to the many other tools currently on the market.

If you are attending OOPSLA, why not take a few minutes to drop by and talk with us? It is always useful to find out what kind of things you're interested in and how you're using Smalltalk. See you there!

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

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

VISIT OUR BOOTH AT OOPSLA!

10 Years Ago, When OTI Suggested That Object-Oriented Technology Would Revolutionize The Software Industry, People Called Us Crazy...

Now, They Simply Call Us.

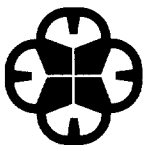
For over 10 years, OTI has been on the leading edge of object-oriented software engineering. And today, as more and more companies adopt this exciting, new technology, OTI remains the leader in providing industrial and commercial object-oriented solutions.

Partners in Object-Oriented Development
OTI's unique technology alliance program provides a means of accelerating product development and introducing new software technology. OTI's technology is being used in products ranging from pen computers to real-time systems. Through these alliances, we've earned a solid reputation for developing high-quality, reliable software – on-time, within budget and to demanding product specifications. This success is attributed to OTI's ENVY®/*Developer* – the first multi-user development environment for object-oriented engineering.

OTI's ENVY/*Developer* – Product Development Tools For Smalltalk
With ENVY/*Developer*, large and small software engineering teams work within an interactive, shared programming environment. Inside this environment, team members share common development tools, common software components and common source code – that means faster cycle times, increased productivity, virtually no duplicated code, and no wasted effort.

Applications are created efficiently and effectively, from beginning to end. Using ENVY/*Developer*, the team passes the application through each phase of the software manufacturing lifecycle – conceptualizing, prototyping, manufacturing, testing, release and maintenance – without ever leaving the environment. ENVY/*Developer* also tracks this process by providing complete software version control and multi-platform configuration management.

Interested?
If your organization is interested in joint research and development or you would like more information on ENVY/*Developer* and object-oriented programming environments, call us today.



Object Technology International Inc.
Engineering Ideas Into Products

Canada Telephone: 613-820-1200 • Fax: 613-820-1202 • E-mail: info@oti.on.ca USA Telephone: 602-222-9519 • Fax: 602-222-8503

- those objects passed in as arguments
- an object's class (by sending the message self class)
- values of instance variables—objects that are part of the object's encapsulated state
- any object returned as a result of sending a message to an object already visible
- objects assigned to temporaries
- any class whose name is known
- you can create an object whenever you need it (assuming you know the name of its class)
- an object that is a value of a global variable (for example, Smalltalk)
- class variables of the object's class or any of its superclasses
- variables in pools specified by the object's class
- constant objects known to the language (e.g., nil, true, and false)
- literals (including integers and floating point objects, strings, literal arrays, a literal block)

Enough! I asked my colleagues for additions and got several that were far too obscure to include in this column. Let's organize these objects into four categories:

1. *Globals of varying scope.* We can include globals, pools and pool variables, and even class variables in this category. These global spaces typically contain objects visible to many other objects. If you can name an object in one of these global spaces, it's yours for the accessing.
2. *Objects that dynamically become known within the context of a method.* These include objects passed in as arguments and any object returned from a message. An object that becomes visible in this way can always be retained for later reference or discarded as needed.
3. *Objects that are part of an object's encapsulated state, i.e., instance variables.*
4. *Basic programming constructs.* It's difficult to write any significant code without using nil, true, or false. Literals also fall into this category and are just as ubiquitous.

EXAMINING THE EXPLORATORY DESIGN

Most collaborations are recorded between objects at the same or next layer of detail. If a designer has figured out the details of an algorithm, quite a number of collaborators at very different conceptual levels may be listed. This is an exception rather than the rule; it is more common to have a vague idea that some kind of collaborative effort is required. Most often collaborators are a list of objects that will become known dynamically, not those that are permanently visible.

Lists of collaborators certainly aren't exhaustive or very precise. But this doesn't mean we have a bad design, just a preliminary one. During the early design stages, we determine when to use the services of some key collaborators; we don't yet need to determine precisely how we will use them. First, we develop a model of what an object should do along with a vague idea of some of its key collaborators. Next, we need to try out a number of alternatives.

To add precision, we need to determine whether an ongoing dialog will be required or whether a single message will do. We need to construct a model of how each responsibility will be accomplished. This requires experimentation, since there's no one right way to decompose a solution. However, when working out these details, there are a number of principles worth following to make your implementation cleaner.

LIMIT VISIBILITY

One guiding principle is to make objects visible to each other on a need-to-know basis. An even stronger statement: Don't retain visibility of any object if you absolutely don't have to. In general, design objects so they know as few other objects for as short a time as possible. If an object only needs to know about another for the duration of a method, pass it in as an argument and let the client supply necessary information. Carrying this to extremes, however, will result in objects with poorly designed interfaces.

SIMPLIFY COLLABORATION SEQUENCES

Complex message protocols that have lots of arguments or require exacting sequences of messages between client and server make objects difficult to use and understand. A balance must be achieved between exposing too much complexity and giving enough controls to the client. Simple interfaces are worth striving for.

For example, I prefer to drive a car with a manual transmission because of the extra control I have, while my mother has driven an automatic car for years. She switched from manual when automatic transmissions became popular because she preferred the simplicity. It certainly is much easier to accelerate a car by sending the single message myCar accelerate. I go through this sequence whenever I need to shift gears before accelerating:

```
myCar depressClutch
myCar shiftGear: a GearValue
myCar releaseClutch
myCar accelerate.
```

Most people prefer a simpler interface, provided the necessary services are offered. Too many software engineers offer a manual transmission when their clients prefer the simpler driving method.

continued on page 11

PRODUCT REVIEW

Jan Steinman and Barbara Yates

Object Technology's ENVY Developer

THE PROBLEM

Since the mid-1970s Smalltalk has been the development environment by which all others are measured. The simple, rapid hypertext-like browsing of code combined with incremental compilation raise programming expectations to the level of instant gratification.

Smalltalk gained a reputation as a toy, not because it lacked power or expressiveness, but because few large systems were written in it. Although it was certainly possible to do big projects in Smalltalk (Smalltalk itself being the best example), most of its work reached a certain critical mass then stopped—roughly at the limit of what one person could manage. The ultimate individual software development environment was just that: an *individual* environment.

A big part of Smalltalk's instant gratification is the way it manages change. Each time you save a method, its source code is recorded in a file and can be retrieved if necessary. This works fine for individual developers, but is unmanageable for teams.

At Tektronix Laboratories we realized that the lack of team facilities was holding Smalltalk back. Tek wanted to reap the object-oriented benefits of Smalltalk on larger projects, so we developed different team programming environments for use within the company. These "groupware" environments fell into two general categories: those that maintained the basic Smalltalk "what you *saved* is what you get" philosophy, and those that followed the C/UNIX "check-in, check-out" philosophy. Beyond this philosophical split, they all attempted to address a common set of basic groupware needs.

NEEDS

We've studied and worked on the groupware problem at Tektronix and as consultants. Through interviews with users and their managers, literature research, and personal experience implementing and using many groupware tools, we came up with a basic set of requirements for Smalltalk team programming, roughly prioritized by importance:

- **Integration.** Groupware must support the combining of code received from different developers, which is primarily a function of detecting conflicts and managing dependencies.
- **Code sharing and concurrency control.** A developer must be able to work on a code module without undue concern that other developers are also modifying the same module.
- **Revision history.** Different versions of code need to be maintained so that if new versions are found to have problems, old ones can be easily retrieved.
- **Configuration management.** Different combinations of code modules need to be assemblable; previous versions of configurations are necessary for regression testing.
- **Documentation.** In addition to standard Smalltalk method and class comments, the new components necessary to groupware require documentation support.
- **Branching and merging.** It is sometimes necessary to diverge from a single development path; then the two paths usually must be brought back together.

Aside from these basic needs, a number of specialized needs are often provided by groupware environments, including performance monitoring and tuning tools, object storage mechanisms, and facilities for generating link libraries. We'll examine how Object Technology International's (OTI) ENVY/Developer, referred to here simply as "Envy," meets these needs.

ENVY PHILOSOPHY

It is apparent that Envy was *designed*, and not simply cobbled together.

Envy adheres fairly well to the philosophy that "few concepts, rigorously applied" are better than special cases for everything. Although it has a complicated user interface, and does take some learning, most users find it predictable and easy to understand once they have absorbed the central concepts.

Envy maintains the original "what you saved is what you get" paradigm, rather than succumbing to the easier-to-implement "check-in, check-out" pattern, and uses the Smalltalk method as the smallest unit of code sharing. This means that team members can instantly view each other's work, fostering communication and avoiding needless branching.

Envy is conservatively designed to avoid accidents. It uses *error avoidance* rather than error detection. If an operation does not make sense in the current state, its menu selection is disabled. Sometimes this can be frustrating, but we're convinced it is much better than picking up the pieces after inadvertently selecting a "you asked for it, you got it" operation. As a corollary to error avoidance, Envy uses multiple browsers to

let you examine the present state of the system rather than rely on multiple reports to tell you what happened after a problem.

Large-scale design is fostered by partitioning the problem into functional units. In fact, Envy's base image comes pre-partitioned into functional units, making it easier, for instance, to substitute a completely different user interface.

Class ownership has been debated in this and other publications. Envy is subtly different. It insists upon class *definition* ownership: Any number of developers can provide methods that extend a class, but only one developer is allowed to change a class's structure. Other groupware systems eschewing class ownership can result in many conflicting definitions for a class, which is deadly to large projects!

Finally, Envy obeys Einstein's dictate that "everything should be made as simple as possible, but no simpler." Where it makes sense to override a concept with a special case, Envy does so.

ENVY CONCEPTS

Envy works from these basic concepts:

- All source code resides in a shared repository.
- There is a hierarchy of software components that have container relationships to each other.
- Loading and unloading a component is atomic.
- Software components progress through stages, from edition to version to release.
- Work in progress is carried out in mutable editions of components.
- Components become immutable when declared versions.
- Users are associated with components in specific roles, which may or may not be enforced.

Shared repository

All source code resides in a shared repository, which accepts changes and makes them immediately shareable. Instead of the typical sources and changes files, images are connected to a shared network repository. As soon as a change to source code is saved, the new code is appended to the repository. Since all the team members are connected to the same repository, code changes are immediately accessible to other members of the team, who can view or load the new code into their image as desired. Both the source strings and the compiled bytecodes are stored in the repository; loading compiled code from the repository is five to ten times faster than file-in.

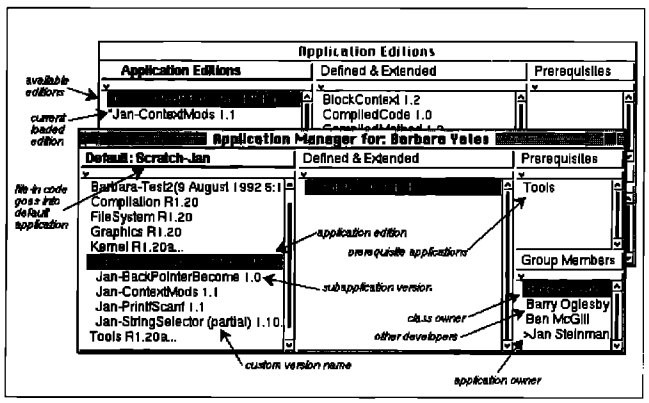


Figure 1: Application manager details, and history of an Application.

Hierarchy of software components

There is a hierarchy of software components that have container relationships to each other. These components are methods, classes, subapplications, applications, and configuration maps. The smallest component is the *method*, which is always a part of a class or class extension. Methods have version history, as do all other components.

Classes differ from *class extensions* in that classes include the class definition and class comment, while class extensions include only methods. Classes and class extensions are contained by applications or subapplications. As mentioned earlier, class extensions provide for multiple owners of bits and pieces of a class. We use the term *class* to mean either class or class extension, unless a distinction is needed.

An *application* is a collection of classes that together serve a useful purpose. Applications declare *prerequisites*, which are other applications required to be present so they can function. Loading an application loads its contained classes and their contained methods (Figure 1).

Applications are actual Smalltalk classes and, as such, they can implement behavior. For example, when an application is loaded into an image, it is sent the message loaded. The developer puts into the loaded method any needed initializations that should occur when the classes in the application are loaded, such as initializing pool dictionaries. Another behavior of applications is that they can respond to some standard system events, such as image start up and shut down, by implementing the methods startUp and shutDown. Objectworks Smalltalk has a similar function via dependents, but since it is implemented using a dictionary, the order of events is nondeterministic. In Envy, system event messages are sent in prerequisite order, so applications can respond to the events in a predictable sequence.

Subapplications are applications with some restrictions placed on them. They are always contained in and loaded as part of an application; they cannot be loaded by themselves. Subapplications have two typical uses: to isolate platform dependencies and to organize classes within a large application. When an application is loaded, the loading of each subapplication is controlled by a boolean configuration expression; that is how a platform-specific subapplication is loaded appropriately. We use the term *application* to mean either application or subapplication, unless stated otherwise.

Configuration maps are named collections of applications. Most teams will use a configuration map to periodically rebuild their image, bringing in the latest integrated and tested versions of all their applications. Another use of configuration maps is a "one button" way to load an application and all of its prerequisites. In a large organization that promotes firm-wide compo-

nent reuse, configuration maps are used to load all the firm-specific versions of base applications, such as those containing Object, String, etc. Other configuration maps are centrally managed to load the latest versions of the firm's reusable components. Each project team may then have its own configuration map to load its applications on top of the firm's customized base, plus whatever reusable components the team needs.

Atomic loads

Loading and unloading a component is atomic. Envy performs "loadability" tests before beginning the load of a component and notifies you of the first error it finds (if any). The image is never left in an inconsistent state—loading either succeeds completely or fails completely. This is especially important in big components, subapplications and larger.

A totally foreign concept to Smalltalk users is that of *unloading*. Any component that has been loaded can be unloaded. Until Envy, a developer typically unloaded unwanted code by ditching the image and file in everything except the unloaded code!

Component stages

Software components progress through stages, from edition to version to release. Work in progress is carried out in mutable editions of components. Declaring an edition to be a version disables changes. A version is released to its containing component. All components make one or more passes through a change cycle between "first code" and completion. Any new component is an *edition* when it is created. Editions can be changed and are signified in the user interface with a timestamp next to the component name. The developer works on the component until it has reached a stage that should be "frozen" (especially if it's working and the developer wants to make some changes that could break it!). The developer then makes the component a version.

Versions are identified by a label next to the component name, instead of the timestamp that denotes editions. Envy suggests version names, but the developer can specify an arbitrary string, such as "for testing 1.0." Once a component has been versioned, it and its label are frozen and cannot be changed. Therefore, before a component can be versioned, all its parts (and all their parts, recursively) must have been versioned.

If developers wish to make changes to a version, they create a new edition of the component. If those changes destroy the component beyond all recognition, or if the developer simply wants to do regression testing, the old, unchangeable versions can be reloaded easily.

At some point, the developers decide it is time to foist their creation on their peers. If they own the component, they can *release* it to its containing component, at which point those who load the containing component get the new part.

To avoid unnecessary interference with the traditional Smalltalk programming style (as well as interference among team members), special rules apply to some components' progression through the change cycle:

- Methods are always editions and, if currently loaded, are implicitly released to their containing class.
- Changing a method in a class version automatically creates a new class edition.
- Classes must be versioned to be released to their containing application or subapplication.

These exceptions allow you to use Envy transparently for at

least 95% of what a Smalltalk developer normally does, while keeping your "work in progress" from being accidentally propagated.

User roles

Envy users fill roles with respect to software components, owners, and developers; flexible access protection may restrict the roles an individual user may fill. The creator of any component automatically has the most authority. This user is called the owner or manager of the component, and can reassign this role to another user. The roles exist for one version and are carried over into new editions until they are reassigned. We use the term *owner* to mean either owner or manager, unless otherwise stated.

Any number of *developers* may be assigned to a class. These developers make changes to the class in their own edition, which they alone can version. The class owner can then release the class to its containing application.

Flexible permissions are associated with an application. Unless the owner of an application explicitly changes it, anyone has permission to load applications, make new editions of classes, and view source code. This default allows development of a class to be a collaborative effort. If desired, application owners can restrict these operations to either themselves or the assigned developers. Private methods can be controlled separately from public methods, enforcing the "contract" interfaces between teams.

Owners of applications and configurations are the only people who may version them. They also have other responsibilities, such as determining the prerequisites for an application or creating new editions. In the simplest case, common in many organizations, one person owns all the classes and manages the application.

ENVY TOOLS

Envy has a variety of browsers for different purposes. Usually the developer will use the Application Manager and one of the development browsers; the choice of browser depends on their preferred view of the "world" of their image. Many operations are available in more than one browser, so the developer is not forced to switch browsers to perform common tasks.

The development browsers consist of two views of the image world: class-centered or application-centered. The Classes Browser arranges all classes in inheritance order and has a second pane that shows which applications define or extend the class. Italics indicate prototols that are not part of the selected application or applications. Many list panes throughout Envy allow multiple selection—doing this in the protocols pane shows the union of their lists in the methods pane. Also available is a Class Browser for browsing a single class.

The Applications Browser (and the single application Application Browser) presents the alternate, application-centered view. Selecting an application shows a list of all the classes it defines and extends, plus a toggle option to show all the prerequisite classes.

Some prefer the Classes Browser and others the Applications Browser. Smalltalk-80 users may find applications somewhat

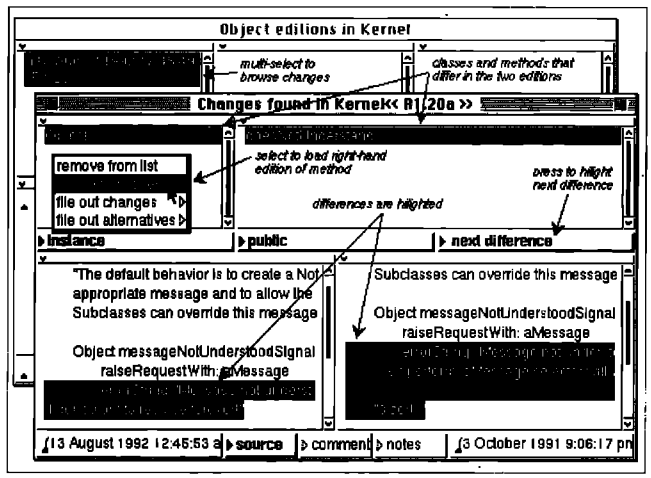


Figure 2: History of Object, showing differences between two editions.

analogous to class categories and therefore prefer the Applications Browser. Smalltalk/V users are often more at home with the alphabetical/hierarchical view presented by the Classes Browser.

The Application Manager allows manipulation of the development stage of applications and classes. This browser lists all of the applications loaded in the image, with subapplications indented according to their nesting level. With one application selected, the other panes list the defined and extended classes, the application's prerequisites, and the application owner and assigned developers. This browser is used for organization and management beyond normal code development, such as loading and unloading applications or classes, versioning applications and classes, releasing classes, and determining the composition of applications.

Recreating an image in Envy is easy. Using the Configuration Maps Browser, simply load one or more configuration maps into the image supplied by OTI. Generally, teams define configuration maps that list the various applications comprising their "deliverable." All of the base image applications in the repository supplied by OTI are already listed in the supplied configuration map called Envy/Manager. Developers can examine existing maps in the repository, create new maps, and edit the contents of map editions. When all the applications in a configuration map are versioned and the map is loaded, a configuration map owner can version it. The map owner does not have to experiment with the load order of the applications in a map—the applications' prerequisites determine the order and the entire load is atomic.

A prime feature of Envy is the collection of tools for version history and comparison. In all the development browsers, it is possible to open a browser on all editions of a selected component. These history browsers list, in reverse chronological order, all the editions and versions of the component. From the editions list it is possible to load a selected edition or select any two editions and browse their differences in a Changes Browser. This browser displays differences by highlighting lines and allows loading of the alternate edition if desired (Figure 2).

Sometimes there will be concurrent development of the same component by two (or more) developers. This happens at the class level because, unlike "check-in, check-out" systems,

there is no exclusive locking of a class to prevent others from making needed changes. This might occur at the application level when a production version of an application is undergoing maintenance while other developers are working on "the next release." The same Changes Browsers that show you the differences between two editions also allow you to merge two editions by installing one or the other version of a method or definition, or by copying, pasting, and compiling a new developer-merged edition of a method

There are two buttons in development browsers worthy of special mention. The public/private toggle displays public or private classes (in the classes pane), or public or private methods (in the methods pane). Private classes should not be referenced and cannot be subclassed outside of their applications. Subclassing of private classes is strictly enforced; referencing results in a warning. Private methods should not be called outside of their inheritance hierarchy. The application owner can deny non-group members the ability to read the private code.

If you don't like the tools provided, keep in mind that Envy is an open system. Certain low-level code that accesses the database is hidden, not so much because OTI doesn't want you finding out their secrets (determined Smalltalkers will find ways to view this code), but because changing these methods could damage the database. Custom user fields can be associated with any Envy component if additional state is needed for some reason. If an organization needs custom capabilities, adding them to Envy is not much more difficult than adding them to Smalltalk. An added advantage is the many reusable classes that can be used royalty-free in your application.

FEATURE COMPARISON

Table 1 shows how some groupware environments compare in solving basic needs of the Smalltalk development team, along with the platforms supported by each. Not all are currently available; we listed those we know about to contrast different capabilities and demonstrate the growth in the genre.

Ad hoc refers to individuals working in separate images, filing out bits of code. This is, unfortunately, how a lot of team Smalltalk is still written.

Change set refers in general to techniques that exploit the Smalltalk–80 change set mechanism. Tektronix developed browser support for multiple change sets; Knowledge Systems Corporation later refined the concept and marketed change set tools.

Team tools, developed for internal use at Tektronix, combined change set tools with configuration management, method revision history, and limited merging. Team tools used UNIX RCS to implement-check–in, check-out concurrency control.

Instantiations enhanced and extended the team tools concepts to produce a product called Application Organizer. Digitalk has since acquired Instantiations; the future of former Instantiations products is unclear.

AM/ST is a Coopers & Lybrand product currently available for Smalltalk/V only. AM/ST was reviewed in THE SMALLTALK REPORT, March/April 1992.

WHO CAN BENEFIT

Not every Smalltalk development team needs a groupware product as powerful as Envy. In particular, teams of up to three people working in the same physical location can get by with ad hoc methods. Corporations with multiple two- to three-person Smalltalk projects can choose to "roll their own," and develop and maintain groupware based on change sets or other file-outs and RCS or SCCS. However, these methods break down as the number of team members climbs above three or multiple teams need to share company-wide reusable components.

Envy really shines for managing large projects with dozens of developers. By spreading project responsibility over three distinct levels (configuration, application-subapplication, and class), managers can control a large project with precision. Since subapplications can be nested, project responsibility can be further divided to an arbitrary level.

Envy has special abilities—as well as an established track record—in developing embedded systems. Anyone wishing to run Smalltalk from anything except a graphical workstation should consider Envy the only solution at this time.

Envy eases parallel development with its merging and differencing capability. Very few projects have the luxury of never needing to split the development path, perhaps for an important demo or due to geographical distance. It is never fun merging diverged code, but Envy makes it much easier.

In short, if you have between roughly 4 and 40 Smalltalk developers on a single project, you can benefit from Envy. The larger the team, the greater the benefit. As the project leader of a successful commercial product using embedded Smalltalk and about two dozen developers put it, "We could not have done it without Envy!"

IMPROVEMENT OPPORTUNITIES

Envy has a solid, industrial-strength feel to it. When something unexpected happens, you tend to question yourself, rather than Envy. It is truly a product without glaring deficiencies; in this case, "improvement opportunities" is not just a euphemism for bug fixes! There are, however, some areas in which OTI should concentrate future development. These are listed in what we believe to be order of importance.

Multiple libraries

While Envy nicely satisfies an unprecedented groupware population of up to several dozen developers working in a single library, it begins to show stress as that number is pushed above 50 or so, or if the organization wants a multi-library architecture. The needs of a corporate-wide code repository are fundamentally different from those of groupware development: ease of finding and browsing functional units predominate. While Envy has export/import ability between libraries, it would be an advantage to be able to access at least a descriptive comment about applications in other libraries prior to importing them.

Renaming and deletion

Renaming is not supported, so you cannot correct mistakes as silly as misspelling an application name. Nor can you delete a version, such as one called OBSOLETE! DO NOT USE! (However, knowing their mistakes will continue to embarrass them tends to make developers more careful!) Envy needs a carefully controlled renaming and deletion facility.

Table 1. Comparison of groupware environments.

	Platforms					Features								
System	80	286	Mac	PM	Win	int	share	hist	cfg	doc	diff	perf	DLL	obj
ad hoc	●	●	●	●	●	①	❶	○	○	○	○	❼	○	❶
change set	●	○	○	○	○	①	❶	○	❶	○	○	❸	○	○
team tools	●	○	○	○	○	①	❸	●	●	○	●	❸	○	○
Object Master	●	○	○	○	○	①	❷	●	○	●	●	❸	○	❶
AM/ST	○	○	●	●	●	②	❺	○	○	●	○	●	●	○
Envy	●	●	○	●	●	●	❹	●	●	●	●	●	●	●

80: Smalltalk–80
286: Smalltalk/V-286
Mac: Smalltalk V/Mac
PM: Smalltalk V/PM
Win: Smalltalk V/Windows

int: dependencies, detecting conflicts
share: code sharing, concurrency
hist: revision history
cfg: configuration management
doc: documentation support
diff: differencing & merging
perf: performance tuning
DLL: link library generation
obj: object storage facility

○ feature is not supported
● feature is supported
① "check with system" available standard with Smalltalk–80
② "off-line" conflict detection possible after load of conflicting code; no dependency mechanism
❶ code sharing and configuration in arbitrary units as decided by developer, with no concurrency control
❷ code sharing and configuration in arbitrary units as decided by developer, using Unix RCS
❸ profiling standard with Smalltalk–80
❹ Binary Object Storage Service available for Objectworks Smalltalk–80
❺ code sharing only at the application level
❻ code sharing of configurations, applications, subapplications, classes, and methods

User interface

Just as climbing a hill reveals the mountain behind, user interface advances bring out issues that other less capable tools have yet to conceive. Error avoidance in Envy is wonderful, but with it comes the responsibility of informing the user what is happening. New users suffer what we call the "gray blues"—wanting desperately to perform some menu item, but being frustrated because the menu item is grayed out (disabled). Context-sensitive help would be a desirable addition.

The need to support so much functionality combined with the need to support multiple plat-

forms creates multiple browsers that differ significantly from the Smalltalk vendor-supplied browsers. (Smalltalk/V users complain about the "Smalltalk-80–like browsers," and Smalltalk-80 users complain about the "Smalltalk/V-like browsers," but they are both complaining about the same browsers!) We can't offer easy solutions, but keeping closer to native browsers would help.

Peer review

An important aspect of successful large projects is peer review. Since browsing others' code is so easy, we experimented with using Envy for code review, as have others (see "Implementing Peer Code Reviews In Smalltalk," S. Sridhar, THE SMALLTALK REPORT, July/August 1992), by making annotations in place. Although it works fairly well with no deliberate support, its usefulness could increase if more attention were given to peer review. For instance, automatic notification that a review had taken place, release controls until review conditions are met, and easy feedback to reviewers.

Documentation

The Envy manual is accurate and concise, but it is only a reference manual. The menu item in each pane of each browser is described in turn, but there is no user-centered, task-based description of the development process. Desperately needed are a tutorial and a "cookbook" of "how do I . . ." questions and answers.

CONCLUSION

Smalltalk groupware has had a long struggling childhood. The recent availability of several products designed to foster groupware ushered in a gangly, clumsy adolescence, with bits missing here and bugs hiding there. Envy brings Smalltalk groupware into adulthood, with a complete feature set that fulfills today's groupware needs and the stability expected of a mature product. Any team of Smalltalkers working on a common project should consider it a leading candidate for solving most programming problems. ■

Jan Steinman and Barbara Yates are partners in Bytesmiths, a technical services company specializing in object-oriented design, implementation, and training. Jan has worked with Bytesmiths' clients to create windowless ("headless") Smalltalk servers using Envy and has conducted evaluations of Smalltalk groupware products for clients. Barbara teaches Envy training classes for Bytesmiths' clients and has assisted numerous teams in conversion to Envy. Together, Jan and Barbara have worked with over 80 Envy users and an equal number of other Smalltalk groupware environment users.

OBJECT VISIBILITY *continued from page 4*

STORE FACTS IN ONE PLACE

If the same objects are used in a number of methods, hold on to this shared information in the object's class. Class methods can be easily designed to yield this default information. It is a matter of style whether these objects should be returned from class methods or stored in class variables. From an instance's perspective, maintenance of this constant information is an appropriate responsibility of its class, regardless of how it is accomplished. This eliminates sprinkling the same literal objects over a number of instance methods. If a literal value needs to be modified, the programmer only has to make the change in one place.

Work at reducing the number of objects that a class depends on. Direct reference to any global objects is considered harmful by many Smalltalkers. Code with "hard-wired" references to other objects is fragile and highly dependent on correct context being established before it can run. It is difficult to reuse code containing global references in another context. To be reused, code must either be reworked to remove direct global references or scaffolding code must be executed to set up the necessary global context.

LIMIT DEPENDENCIES ON OBJECT STRUCTURE

Sending messages to self is a valuable implementation technique for two reasons: It allows programmers to separate detailed steps from main parts of an algorithm, and clearly identifies steps in an algorithm that can performed differently by a subclass method.

Just as important, sending accessing messages to self allows code to be insulated from changes in instance variable structure. It also allows subclass developers to override those accessing methods and provide the necessary information in another way.

DEVELOP A SENSE OF STYLE

Don't try to use every language construct when translating design-level collaborations into a Smalltalk implementation. Current Smalltalk environments have too many ways, for my taste, to make objects visible. Teams should develop and stick to a style guide that addresses when and how to use particular Smalltalk constructs and how to simplify collaboration patterns. Smalltalk programming style is an art and different organizations quite naturally develop their own styles. It is important to cultivate a sense of style and create some coding guidelines before translating a design into code. ■

Rebecca Wirfs-Brock is Director of Object Technology Services at Digitalk, co-author of Designing Object-Oriented Software, and program chair for OOPSLA '92. She has 17 years' experience designing, implementing, and managing software products. For the last eight years she has focused on object-oriented software, including managing the development of Tektronix Color Smalltalk and developing, teaching, and lecturing on object-oriented software. Comments, further insights, or wild speculations are greatly appreciated by the author. Rebecca can be reached via email at rebecca@digitalk.com. Her U.S. mail address is Digitalk, 921 S.W. Washington, Suite 312, Portland, Oregon 97205.

GUIs

Greg Hendley and Eric Smith

Separating the GUI from the application, Part 2

In a recent column (THE SMALLTALK REPORT, May 1992) we presented an application architecture for separating the host-GUI–dependent, presentation-dominant parts of an application from the control and semantic portions. In brief, the ICM architecture divided interactive applications into three primary components: (1) the interface component, responsible for all aspects of input handling and output presentation that directly involve host GUI features; (2) the control layer, the actual intelligence, which carries out commands, maintains selections, keeps track of operational validity, etc.; and (3) the domain model layer, comprised of all the objects representing the information with which the user is working.

We pointed out several advantages deriving from the use of this architecture. Chiefly, ICM applications are very easily ported to different platforms. Second, maintenance is eased because less volatile sections of code are insulated from more volatile ones like the interface. Finally, project maintenance is facilitated because the work of application developers and user interface specialists is more clearly delineated.

SCALING UP

The example code we previously provided implemented a simple log-on dialog. Although it illustrated the concepts of ICM, it was much too simple to be a useful guide to implementing an entire application. We will try to make up for this by covering in some detail a few of the problem areas that arise when one first attempts to construct an ICM application.

GUIDING THE USER

In any reasonably modern GUI-based application, end users are likely to expect menu selections and push buttons that represent currently invalid operations to be disabled or grayed out. Under the ICM model, implementing this behavior involves both the interface and the control components.

The control part of the application knows what each command's prerequisite conditions are. If well designed, it knows immediately when any given command has become invalid. However, it has no knowledge of what type of user interface element presents the command as an option to the end user. It is possible that the current interface does not present the command at all. Therefore, the control must pass on to the interface a request to disable whatever interface element, if any, it uses to present the command in question.

For example, assume that the interface has two buttons A and B, which represent the control commands cmdA and cmdB, respectively. Further assume that when one is pressed, the other becomes invalid. In Smalltalk/V PM, this would result in the code segment shown below.

INTERFACE CODE

The following four methods are instance methods of some subclass of ViewManager.

```
bcA: aPane
    "The end user pressed button 'A'."
    self control cmdA
bcB: aPane
    "The end user pressed button 'B'."
    self control cmdB
    disableCmdA "Command A is no longer a valid option."
    (self paneNamed: 'buttonA')
    disable disableCmdB "Command B is no longer a valid option."
    (self paneNamed: 'buttonB') disable
```

CONTROL CODE

The following two methods are instance methods of the class that defines the control for the interface.

```
cmdA
    "The end user has chosen command A."
    "Do whatever needs done to the domain model here."
    "Command B is no longer an option."
    self userInterface disableComandB
cmdB "The end user has chosen command B."
    "Do whatever needs done to the domain model here."
    "Command A is no longer an option."
        self userInterface disableComandA
```

This may seem like quite a few methods just to accomplish a simple task. However, there are many advantages to this approach. First, if the user interface experts later decide that these commands should be represented both with buttons and menus, only the interface layer would need to change. Each of the disable messages would then disable both a button and a menu option. The control layer would remain unchanged, unaware of whether it is disabling a button, a button and a menu selection, or nothing at all as a result of sending the message to the user interface.

Another advantage is that portability has been maintained. Should the application be moved to another platform that uses a different protocol for disabling user interface elements, then

CALENDAR

November 16–20
C++ World
Meadowlands Hilton, NJ
212.274.9135

February 1–4, 1993
Object World (OMG)
Boston, MA
800.225.4698

February 1–4 and
February 4–5, 1993
OOP'93 and C++ World
Munich, Germany
212.274.9135

February 21–26, 1993
Software Development '93
San Jose, CA
415.905.2741

March 8–11, 1993
XWorld
New York, NY
212.274.9135

March 8–12, 1993
INTEROP
Washington, DC
800.INTEROP

all that must be reimplemented is the interface layer. There is no need to comb through the control code looking for protocol that depended on the old host GUI.

QUICK POP-UPS

A small but tempting violation of the separation between the platform specific interface and the portable control occurs when very small amounts of additional information are required when carrying out a command. A user may need to obtain a file name or confirm an unexpected or extreme consequence. In that case, the quickest solution is to have the control ask the user directly. For example:

```
cmdA
    "The end user has chosen command A."
    | fileName |
    fileName := Prompter prompt: 'Enter a file name' default: 'file.dat'.
    "Do whatever needs done to the domain model here."
    "Command B is no longer an option."
    self userInterface disableComandB
```

However, this presents several difficulties. By referring directly to the class Prompter, platform-specific information is woven into the application control and cross platform portability is compromised. Second, if the user interface designers decide to use some dialog other than the prompter to obtain file names, then all of the control layer must be examined for expressions such as those above. Finally, the user interface may have already obtained a file name from the user, which was entered in a text entry field in the window from which this command was initiated. The control code does not and should not know which is the case.

Although tedious, the best solution is to go to the user interface to accomplish all tasks. The control should send a request to the user interface to obtain the file name by any means and return it. The following two methods illustrate this.

Interface code

```
getFileName
        "Sent by the application control.
        Answer a file name or nil if one is unavailable."
    | aFileName |
    aFileName := Prompter prompt: 'File:' default: 'FILE.DAT'.
    (aFileName isNil or: [aFileName trimBlanks isEmpty])
        ifTrue: [^ nil].
    ^ aFileName trimBlanks
```

Control code

```
cmdA
    "The end user has chosen command A."
    | aFileName |
    aFileName := self userInterface getFileName.
    "Do whatever needs done to the domain model here."
    "Command B is no longer an option."
    self userInterface disableComandB
```

This problem worsens when an untoward event discovered deep within the domain model—the worst possible place to directly involve platform-specific classes—requires a confirmer or quick dialog. The best solution is to use some kind of exception handler so that the domain model code can notify the control code of the unexpected problem. The application control assisted by the user interface can help manage the necessary interaction with the end user.

SUBORDINATE APPLICATIONS

It is problematic to mix interface and control code when a secondary application is opened as a result of a user command. The application control recognizes the need to open a new window but knowledge about the protocol used to open new windows, and the classes that represent them, is interface- and platform-specific and should not be included in the control layer. The following method in the control layer violates the ICM architecture.

```
cmdBrowseDocument
"The end user wants to open a browser on the selected document."
    SomeKindOfViewManager new
        openOn: self selectedDocument
```

This method causes many of the same difficulties as direct reference to class Prompter in the first example above. The decision as to which interface should be used in browsing documents is moved out of the interface layer. Further, this type of reference, from the control layer to a class in the interface layer, introduces a complication when porting the application to a new platform. When there are no direct references to classes in the interface layer in either of the model or control layers, these lower two layers can be easily ported to a new platform and a new interface layer built on top of them. Methods like the one above will introduce unresolved references when ported without the interface layer. These will have to be carefully located and resolved when the new interface is constructed.

Once again, the correct method for handling this sort of problem is to pass the problem back to the user interface in a manner similar to that used for prompters and confirmers. This might result in the following set of methods.

Interface code

```
mcBrowseDocument
    "The end-user has chosen the Browse Document menu option."
    self control cmdBrowseDocument
createDocumentBrowserOn: aFile
    "Open a document browser application on the argument."
    self documentBrowserClass new openOn: aFile
documentBrowserClass
    "Answer the class which defines the preferred document browser
    interface."
    ^ SomeKindOfViewManager
```

Control code

```
cmdBrowseDocument
    "The end user has chosen command A."
    self userInterface
        createDocumentBrowserOn: self selectedDocument
```

Again, this might seem an excessive number of messages back and forth between the interface and control portions of the applications, when one could simply ask the control for the selected document and open the correct kind of browser from the mcBrowseDocument method. However, as the application grows in complexity, the question to browse, or exactly which document to browse, may become quite complicated. Such a decision will involve numerous factors of which only the control layer is aware. If the interface has short-circuited the control's responsibilities the command will behave incorrectly.

OTHER PROBLEMS

As we move up the scale to more sophisticated applications with increasingly rich interfaces, nastier problems begin to crop up. Handling errors and exceptions can prove especially difficult. This can involve sudden invalidation of assumptions made by both the interface and control layers. The designer of the control must be able to provide the user interface layer with notification of any exceptional conditions. The interface must be able to present the end user with useful, non-confusing information regarding the situation. This task is especially difficult without a good exception-handling mechanism.

The separation of presentation and control is most difficult to maintain when the presentation of the underlying domain model to the end user is highly graphical in nature. The most convenient implementation in such cases is to design the model objects so that they know how to draw themselves on some graphic medium. However, this involves burying platform-specific code all the way down in the domain model.

In our experience, there are several plans of attack for solving these problems. Not all of them are entirely satisfying, especially in the case of exception handling. As solutions for these situations evolve, we will include them in future columns. ■

FOR FURTHER READING

Many of the ideas on which ICM architecture is based, particularly the strong separation of presentation and control, grew out of the work of the Dialog Management System group at Virginia Tech in the late 1980s. For interested readers the following references are provided:

Hartson, H. R., Control and communication in user interface management, Technical Report TR 88-3, Department of Computer Science, Virginia Polytechnic Institute and State University.

Hartson, H., R. Johnson, D. Hix, and R.W. Ehrich, A human-computer dialogue management system, PROCEEDINGS OF INTERACT '84, London, England: IFIP, Vol. 1, pp. 57–61.

Yunten, T. and H.R. Hartson, A SUPERvisory methodology and notation for human-computer system development, ADVANCES IN HUMAN-COMPUTER INTERACTION, H. Rex Hartson, editor, Ablex, 1985.

Greg Hendley and Eric Smith are both technical staff members at Knowledge Systems Corp. Greg Hendley's OOP experience is in Smalltalk/V (DOS), Smalltalk-80 2.5, Objectworks Smalltalk Release 4, and Smalltalk/V PM. Eric Smith's specialty is custom graphical user interface using Smalltalk (various dialects) and C. The authors may be contacted at Knowledge Systems Corp, 114 MacKenan Drive, Suite 100, Cary, NC 27511, or at Compuserve 72000,1056.

SMALLTALK IDIOMS

Kent Beck

Collection idioms

Compared with procedural languages, Smalltalk's collections feature is universally regarded as saving the most programmer time. The Smalltalk collection hierarchy has been widely copied by many, including the popular National Institute of Health class library for C++. Along with numbers, collections share the distinction of being the most portable class among the three major Smalltalk implementations: Objectworks\Smalltalk, Smalltalk/V (all flavors), and Enfin/3.

Once I started talking to my friends about how they use collections I realized I had enough material for two idiom columns. Most Smalltalk programmers don't take full advantage of collection's features, but the more experienced have a bag of tricks (some of which are not obvious at first glance) with everything collections have to offer. These programmers also know where the traps lie and how to avoid them.

The remainder of this column takes you through the perils of subclassing collections and some of the richness of the collection protocol. Next month we'll take a brief tour of the most common classes, how they are implemented, and when they should be used.

SUBCLASSING COLLECTIONS

My aesthetic sensibilities are always offended when someone creates a subclass of a collection class just because the object being created includes a collection. The most obvious example of this kind of subclassing is SystemDictionary. Until I started writing this column I never had a solid engineering explanation for my reaction. Now I think I can explain.

Unfortunately, subclassing a collection is one of the first ideas that comes to mind when you finally understand inheritance. "Oh, I need a polygon. I'll just subclass OrderedCollection. That way I'll get all the adding behavior for free." Lo and behold, you can add and remove points from a polygon as soon as you define the class. Pretty neat, this Smalltalk stuff.

It's not until later that the danger of subclassing a collection becomes apparent. While there may be a couple of messages that make perfect sense for your new class, others don't make sense and still others are actually harmful. I confirmed this by executing Smalltalk removeKey: #Object in Objectworks\Smalltalk. Away went class Object, never to return. Smalltalk/V Mac asks for confirmation that you want to delete the class, but there are other messages just as harmful that no one thought to protect.

By subclassing to gain a collection you have opened up an enormous window onto the implementation of your object, violating its encapsulation and potentially opening it up to harmful messages. You can protect your class by overriding the offending methods with self shouldNotImplement. By the time you are done, though, you will have a class that gainfully inherits a couple of messages while explicitly eliminating a dozen others. Even so, you are still vulnerable to someone coming in later and adding a method to the superclass that re-exposes your subclass. At that point you may as well have inherited from Object, added an instance variable for the collection, and forwarded the messages you cared about to the collection.

Back in the olden days, there were few gratuitous subclasses of collections. Objectworks\Smalltalk 4.1 has a half dozen classes that inherit from a collection, but don't otherwise act like collections. In its defense, UninterpretedBytes (more of which later) is a subclass of Object even though it is implemented as a collection of numbers. In looking at the V image I see only CompiledMethod and Process as collection subclasses that don't really belong (both of these classes are done "right" in OW\ST).

This perspective on subclassing collections runs counter to my usual advice on using inheritance. I am a firm believer that inheritance *does* share implementation, and that's what it should be used for. Rather than read inheritance as "is-a" or "is-kind-of," I read it as "is-implemented-like." This explanation of inheritance is simple for beginners to grasp. It admits a simple metric for evaluating inheritance decisions, such as which alternative allows the most code sharing. Beginners can flounder for months trying to understand "inheritance as abstract specification" (*a la* contracts) or "inheritance as classification" (*a la* AI).

I don't have a glib response to this apparent inconsistency. Perhaps the reason collections are not good to inherit from is that they have so much behavior at the abstract level. Any subclass that isn't really a new kind of collection is bound to find many of those methods inappropriate. Perhaps collections have too much behavior and a different factoring of the system would yield a more satisfying answer. I do know that subclassing to share implementation usually works, but that collections are a notable exception to that rule.

INDEXABLE SUBCLASS

While I'm on the subject of subclassing and collections let me mention a life-saving facility I have had occasion to use once or twice. Let's say you followed the above advice and made your objects subclasses of Object and gave each one an instance

variable that holds onto a collection. If the objects are small and numerous, the overhead of the additional object (usually 12 bytes of object header and 4 bytes in the referencing object) can add up. If the collection is simple (an Array, for instance) you can eliminate the space overhead and improve the locality of reference by declaring your object to be an indexable subclass. This will add a number of indexed instance variables (the number is set at instance creation time in the argument to new:) to your object. The conversion will be made much easier if you were careful to use collectionAt: and collectionAt:put: to access the collection. You can convert:

```
collectionAt: anInteger
    ^collection at: anInteger
```

to:

```
collectionAt: anInteger
    ^self at: anInteger
```

and so on.

COLLECTION MESSAGES

Collection implements a variety of behavior for its subclasses. It is a triumph of object design that all of that functionality depends only on the existence of three methods in a subclass: do:, add:, and remove:ifAbsent:. When implementing new kinds of collections, I have been amazed at how quickly I can get going just by implementing those three methods.

ENUMERATION

Of the behavior implemented in Collection, the enumeration methods are the most powerful and hardest to understand. The methods are interesting because they are safe to use: None of them modify the collection they iterate over. The ones that return a collection always allocate a new object for the result.

I'll go through the messages, describing what each one does, how it is implemented, and when you might want to use it.

do:

Do: executes a block for each element in a collection. It operates strictly through side effects and the results of evaluating the block are discarded. Do: must be redefined in each new subclass of Collection.

I went through all senders of do: in the Smalltalk/V Mac 1.2 image and I couldn't find any clever idioms. I was surprised at how often it was used when one of the other messages would have served better. Interestingly, the times do: was used incorrectly were primarily when a temporary variable was experiencing side effects. If an argument or instance variable was changed the use of do: was usually correct. As a positive example, look at Collection>>printOn:

```
printOn: aStream
    aStream nextPutAll: self class name.
    aStream nextPut: $(.
    self do: [:each | aStream print: each; space].
    aStream nextPut: $)
```

collect:

Instead of just executing code for its side effects, perhaps you want to transform all the elements of a collection. Collect: executes a block for each element, but saves the results and returns them when done. For example, if you want to return the absolute values of a collection of numbers you could write:

```
absolute: aCollection
    | result |
    result := aCollection species new: aCollection size.
    1 to: aCollection size do: [:each |
        result at: each put: (aCollection at: each) abs].
        ^result
```

or you could just write:

```
absolute: aCollection
    ^aCollection collect: [:each | each abs]
```

Collect: and the following messages all have the admirable property of removing the need for temporary variables when they are used. Methods often shrink by several lines when you find a way to use one of the enumeration messages.

Another big advantage of enumeration messages is that they are not sensitive to the kind of collection they operate on. The first version of absolute: above assumes that aCollection is indexable by integers (responds to at: and at:put: with an integer first argument). If I decided later that the parameter to absolute: could also be a Set, which isn't indexable, I would have to change absolute: to deal with both cases. Since all collections respond to collect:, by using it instead I am completely insulated from changes in what kind of Collection is passed in .

Here is another example where collect: is useful. I often make the mistake of converting objects several places within a single class. For instance, I might write:

```
foo
    strings do: [:each | each asSymbol ... ]
```

Then I might convert strings to symbols in several other loops in other methods. The object in question isn't taking enough responsibility. It should provide the service of converting its strings to symbols:

```
stringsAsSymbols
    ^strings collect: [:each | each asSymbol]
```

Then I can write:

```
foo
    self stringsAsSymbols do: [:each | ...]
```

What advantages does this approach provide? First, it's more modular. If I want to stop storing strings and store something else (or compute it on the fly) I can just change stringsAsSymbols and not have to touch every method where the instance variable strings was used. Second, if converting strings to symbols is a performance problem I may never see it if it's buried in half a dozen methods. Putting it in a single method makes the performance implications clear and provides a simple way of implementing caching should that become necessary.

GETTING REAL

Juanita Ewing

The dangers of storing objects

Smalltalk systems now include the ability to correctly write representations of composite objects to disk. Early Smalltalk systems could not deal with objects containing circular references, so the capability of storing objects was not widely used. Now that many kinds of objects can be written, other issues have arisen: When is it appropriate to use this mechanism? Is this a good way to provide long-term storage of objects? Can this capability be overused or misused?

Object storage was first implemented for Tektronix Smalltalk by Steve Vegdahl.[1] In Smalltalk/V this capability is called Object Filing. In Objectworks\Smalltalk, this capability is implemented by BOSS (Binary Object Streaming Service).

In all these implementations, an encoded representation of an object is written to a file. The representation of the object consists of structural information required to recreate the object from the data in the file. Objects recreated from the data on disk are not the same as the original object. These systems do not maintain object identity across read/write operations and are therefore not persistent object systems.

WHAT IS WRITTEN TO DISK?

When the representation of an object is written to disk, it must include all the data necessary to recreate the object. The class name is written to designate the class of object to be recreated. Each component of the object, numbered slots and instance variables, is written. If the component is a reference to another object, that object is also written.

Each implementation has different restrictions on precisely which objects are written. The values of global variables such as Transcript are not written. Instead, a reference to the variable's name is stored and when the object is recreated its reference is bound to the current value of the identifier.

The representation of an object in these systems is the data from the private internal implementation of the object. The public interface to an object is not used to recreate the object. Private, low-level methods are used instead.

WHY DO DEVELOPERS WRITE/ RECREATE OBJECTS?

The big advantage of object storage systems is that they permit a Smalltalk developer to externalize objects without designing a special file format or writing input/output methods. Developers might use object storage systems to "transfer" objects from one image to another. Other members of a development team might need an object that is difficult or time consuming to recreate. A prototype might have objects built by hand instead of programatically or objects might be created from a data feed.

Developers sometimes use this ability to "save" objects; they want the objects to exist longer than an image. Another use is to reduce the size of an application image by building an external "database" of stored objects. Then only the objects that are actually being used need to be loaded.

SOPHISTICATED USE

An application I helped develop had visual components that were used off-screen to generate a composite graphic. This graphic was stored in an instance variable, but we didn't want it saved when we wrote our objects to disk. It was large and took more time to read from disk than to recreate. We needed a way to control which components of an object are written.

Both Object Filer and BOSS have a mechanism to customize what is written on a per class basis.

- With Object Filer, you implement a method with the selector fileOutSurrogate:, which returns a surrogate object to be written to disk in place of the receiver. The surrogate can be a copy of the original object with modified instance variables.
- With BOSS, you implement a method with the selector representBinaryOn:, which uses other BOSS methods to write the representation of the object to a stream.

Sophisticated use of these systems requires developers to write special methods that modify the written representation of the object, usually by changing the private instance state of the stored object. The manner in which these systems are customized is an indication of the limitations of these systems; they manage the storage of an object at the structural level.

DANGERS

Class definitions are volatile. Instance variables, class variables, and pool dictionaries can be added or deleted. Once a change is made to the private implementation of an object, such as adding an instance variable, the written representation on disk is no longer accurate. Because the representation consists of private implementation data, the public interface of that object is not used to recreate the object.

Problems arise from: renaming a class; changing representations; restructuring a class; and refactoring a hierarchy.

Most of these systems have mechanisms to handle *simple* variations in an object's definition. In the case of added and deleted instance variables, Object Filer brings up a graphical interface that interactively lets you map instance variables on disk to the instance variables in your image. This mechanism is particularly useful when instance variables have been renamed. Object Filer also has a mechanism to support classes that have been renamed.

CHANGING REPRESENTATIONS

Suppose a composite object consists of a deeply nested tree structure. When this object is written to disk, a representation of it and all its composite objects is written. Later, the developers add a cache of recently accessed leaf node to the object. This cache, an instance of OrderedCollection, is stored in an additional instance variable. The representation of the object on disk does not specify a value for the cache instance variable. When the object is recreated, it has a nil value for the cache.

The methods in the composite object must be specially designed to accommodate a value of nil for the cache. Accessing methods for the cache must check for nil instead of assuming an instance of OrderedCollection and, if necessary, create an instance of OrderedCollection. The developers then save some composite objects to disk.

Later the cache is changed to be an instance of Dictionary. Accessing methods are again modified to check not only for nil, but for instances of OrderedCollection; if necessary, the cache is modified to be an instance of Dictionary. More composite objects are saved to disk.

What is the situation now? The developers now have representations of composite objects with the following variations:

- no cache instance variable
- cache instance variable bound to instance of OrderedCollection
- cache instance variable bound to an instance of Dictionary

In this example of changing representations, what you really have is a mess, with code for backwards compatibility in every relevant accessing method. The situation is even worse if you don't use accessing methods and instead directly reference instance variables. You end up with code for backwards compatibility in every method that references the instance variable.

The series of modifications I've described is very typical. The original definition of a class is rarely correct; definitions are changed to accommodate optimizations as described above. Functional extensions also require modifications. For example, an ellipse class describes an elliptical element with a border width and color. It has instance variables to store the attributes' width and color. The developers later add functionality for filling the inside area of the ellipse. The class definition is modified as another instance variable stores the fill color.

REFACTORING AND RESTRUCTURING

The most devastating kind of change is not addition or deletion of instance variables. It is the refactoring and restructuring of classes into sets of classes, or the combination of several classes into a single class. As developers create an application, the design evolves. Responsibilities are redistributed and new classes are created.

Let's look at a simple example of restructuring. Suppose your application records information about people such as their name, which is an instance of String. Later you decide a single string is not a good representation and you want to model the first and last names as two separate entities. If you have stored objects with the name represented by an instance of String, you must make extensions to the object storage system to:

- Read the name
- Detect the class
- Potentially parse the string to model first and last names separately.

An example of refactoring recently discussed in several publications is from the Objectworks\Smalltalk user interface library. The class View has been refactored into a number of smaller classes, each with less functionality. Is it possible to take a view that has been stored on disk and recreate it in terms of the new classes? No doubt it would be easier and less time consuming to rewrite the code used to create the view than to recreate its equivalent from the object representation on disk.

ALTERNATIVE

It is easier to rewrite code to make a view because rewritten code uses the public interface to objects. Writing objects to disk using the private implementation data is okay for a quick transfer, but not a good idea for any long-term needs.

Object storage systems are very handy for short-term use, but because of the dynamic nature of classes, they are unsuitable for long-term use. These systems encode structural implementation rather than the semantics of information.

Every major Smalltalk application I know of that used an object storage system for long-term storage ultimately had to be modified to use a less implementation-dependent storage format. A good format captures the data without directly specifying objects and the values of their instance variables. Instead it captures relevant data in an object-independent format by storing only semantic data. Methods that read the data instantiate new objects by sending public messages. ■

Reference

1. Vegdahl, S.R. Moving structures between Smalltalk images, PROCEEDINGS OF THE ACM CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES AND APPLICATIONS, Portland, OR, September, 1986, pp. 466–471.

Juanita Ewing is a senior staff member of Digitalk Professional Services (formerly Instantiations, Inc.). She has been a project leader for several commercial object-oriented software projects and is an expert in the design and imlementation of object-oriented applications, frameworks, and systems. In a previous position at Tektronix Inc., she was responsible for the development of the class libraries for the first commercial-quality Smalltalk-80 system. Her professional activities include Workshop and Panel Chairs for the annual ACM OOPSLA conference.

THE BEST OF comp.lang.smalltalk

Alan Knight

Some Smalltalk stuff

The last few editions of this column have dealt with very broad O-O issues. This time we will discuss three detailed, language-specific issues: Smalltalk text, implementing method pre- and postconditions, and determining a source filename during filein. Although we can't solve all of the problems, we will get a better understanding of them.

FORMATTING

There are programs available for formatting or "pretty-printing" most computer languages. The simpler ones, based on recognizing simple syntactic cues, often break when confronted with complex syntax or strings with escape sequences. The more sophisticated a formatter gets, the closer it comes to actually parsing the language.

ParcPlace Smalltalk has a built-in formatter. Because it is part of an integrated environment, it can directly use the parser to do its formatting. This is not necessarily good, as William Eric Voss (voss@cs.uiuc.edu) describes:

> **I generally love the 'format' item on the CodeView menu....However, occasionally I encounter a long method with more than a dozen lines. I would like to place inline comments in such methods. However, if I then invoke 'format' my comments jump a line or more, often becoming very misleading as a result.**
> **Could someone clearly explain why this happens?...Does anyone have a workaround (other than don't use 'format')?**

Danny Epstein (dje@scs.carleton.ca) explains:

> **The 'format' command works by parsing the source code and then pretty-printing the parse tree. When a comment is read in (by the scanner, if I remember correctly), it is attached to the 'current' parse node. This isn't really what is desired since there are several places in the source code where a comment could go, all of which would get associated with the same parse node. Multiple comments are handled, but their positions are not stored. A better technique would be to associate a comment with the parse node whose code immediately precedes it. If there are several, then the largest one should be used. For example:**
>
> ```
> x := 1 + 2. "comment for statement"
> x := 1 + 2 "comment for +".
> ```
>
> **Note that the second comment is not bound to the 2. The pretty printer then outputs comments after the code. The only exception is that comments are never associated with the entire method. What we call method comments are really comments on the method header (since they appear after it). You could change the parser as described above. I can't think of a quick hack to fix the problem.**
> **All this being said, I myself never use 'format' because I don't like its formatting rules. *C'est la vie.* A good formatter should have lots of user options so it can get close to what the user would do manually.**

Unfortunately, this explanation doesn't provide a solution or workaround, only an understanding of the source of the problem. Anybody care to undertake the job of writing a really good formatter for Smalltalk?

ASSERTIONS

One of the nice things about Smalltalk is its flexibility, its ability to implement interesting features of other languages. One worthwhile feature might be assertions, which allow you to specify the behavior of code in a way that can be checked (as opposed to comments). Assertions are a staple of formal methods and an important part of the Eiffel language. It's easy to do a trivial version of assertions. We define an Object method:

```
assert: aZeroArgumentBlock
    self assertionCheckingIsOn ifTrue: [
        aZeroArgumentBlock value ifFalse: [
            self error: 'assertion failed']].
```

We check some sort of state variable to indicate if assertions are active; if so, we evaluate the block. An example of using this method is:

```
someMethod: aParameter
    self doSomeWork: aParameter;
        assert: [alreadyProcessedList includes: aParameter].
    ^self.
```

This verifies that the parameter has been added to the list of processed items. Although this is useful and provides about the same level of functionality as the C "assert" macro, it's not nearly up to the level of Eiffel assertions, which are built into the language. Eiffel supports assertions as method preconditions, postconditions, and class invariants. A precondition specifies the necessary conditions before a method can execute and is checked just before execution. A postcondition specifies what should always be true after the method has finished executing and is checked just after method execution. A class invariant specifies something that should always be true for an instance of a class and is checked every time an operation modifies an instance. We'd like to be able to use these much more useful assertions in Smalltalk. Bernhard Humm (humm@cs.uow.edu.au) specifies the requirements in more detail:

I would like to introduce the concept of ASSERTIONS (e.g., [Meyer 90]) into Smalltalk: pieces of code to be executed before (precondition) and after (postcondition) execution of the method body. I would have thought extending Smalltalk with this feature would be easy. I defined the following requirements:

- The definition of the method body is done in exactly the same way as without using assertions. The semantics of execution does not change (including the semantics of a return statement and a missing return statement).
- Pre- and postconditions are defined in the method definition (not in separate methods).
- Invoking the method with the assertions does not differ from invoking the method without assertions (this ensures that you can add assertions to previously defined messages without changing other parts of the system).

Example:

```
plus: aNumber
 ^self
  preCondition: [aNumber isOfType: Integer]
  body: [^aNumber + self]
  postCondition: [:res| res isOfType Integer]
```

However, the implementation of **preCondition: body: postCondition:** seems to be difficult. The problem is the semantics of the return statement (which, when encountered, immediately exits the method invocation without any chance to perform the postcondition).

This clearly defines the previously described problem with the assertion mechanism. The return statement apparently makes it impossible to be sure assertions will be checked anywhere except the beginning of the method, and even this cannot be guaranteed if there is a return statement in the assertion block.

Blocks

The reason for the difficulty is the peculiar nature of blocks in Smalltalk. Blocks are similar, but not quite identical, to functions (in a language where functions are first-class). Blocks can have local variables (at least in recent ParcPlace implementations); they can be assigned, passed as parameters, and evaluated. They are also lexically scoped: a block "inherits" the scope of the method by which it was created.

Blocks and functions differ in the return statement. A return exits from a function but exits from the method in which the block was defined. This is necessary because of how Smalltalk uses blocks, but it can cause difficulties and confusion. Consider the following collection method:

```
detect: aBlock
    self do: [:eachItem |
        (aBlock value: eachItem)
            ifTrue: [^eachItem]].
```

In this case, we really want the return to exit from the detect: method rather than either of the enclosing blocks. If statements are written using blocks, a return that only exits the local block would make it impossible to write the common Smalltalk statement:

```
someCondition
    ifTrue: [^something]
    ifFalse: [^somethingElse].
```

On the other hand, consider the case of a complicated sort block:

```
someMethod
    | sortBlock collection |
    sortBlock := [:thing1 :thing2 |
        thing1 condition1 ifTrue: [^true].
        thing1 condition2 ifTrue: [^true].
        (thing1 condition3 and: [thing2 condition1])
            ifTrue: [^true].
    ^false].
    collection := SortedCollection sortBlock: sortBlock.
    ^collection.
```

If blocks were really functions this would return a SortedCollection using this peculiar sorting condition. Instead, it returns a collection that reports an error as soon as an item is inserted. Specifically, someMethod returns the local collection. If we then say:

```
collection add: anObject.
```

the same invocation of someMethod tries to return again, causing a very confusing walkback. The very idea of a function invocation returning twice is bizarre.

Different semantics don't cause a serious problem in this case, which is easy to work around. We can implement a method to do the comparison, or it can be written using nested ifs or a case statement. Complex code inside blocks present one reason I find the lack of any kind of case statement in Smalltalk irritating enough to write my own. These semantics cause more difficulty for assertions.

Back to assertions

One way of handling the problem would be to define two separate kinds of return operations, one restricted to blocks. This would do the job but is a lot of work, a substantial change to the language, and hardly fits the description of Smalltalk as being flexible enough to easily implement language features.

Fortunately, at least in ParcPlace Smalltalk, there is an easier way, which Mario Wolczko (mario@cs.man.ac.uk) describes:

> In Smalltalk-80, since version 2.4, you can associate an 'unwind' block with a method to deal with exactly this situation. Example:
>
> ```
> [f := (Filename named: 'foo') writeStream.
> self doSomethingWith: f]
> valueNowOrOnUnwindDo: [f close].
> ```
>
> Even if the code invoked by **doSomethingWith:** causes a return 'over' this method, the 'unwind' block (argument to **valueNowOrOnUnwindDo:**) will be executed, closing the file cleanly.
>
> Your method will look something like:
>
> ```
> preCondition: preBlock body: bodyBlock postCondition:
> postBlock
>  self check: preBlock.
>  ^bodyBlock valueNowOrUnwindDo: [self check: postBlock]
> ```

This seems an ideal solution to a very difficult problem. My only question is whether there might be a substantial performance cost associated with using an unwind block.

FINDING FILENAMES

Another question from William Eric Voss (voss@cs.uiuc.edu):

> When you have a multiple file goodies package, it is very common to have a file which looks something like:
>
> ```
> | baseDir |
> "Change the next line then fileIn this file."
>  baseDir := Filename named:
> '/where/this/stuff/lives'.
> (baseDir construct: 'file1.st') fileIn.
> (baseDir construct: 'file2.st') fileIn.
> ...etc...
> ```
>
> 'Less portable implementations use **string1**, **string2** instead of the **construct:** method.'
>
> It seems like there should be some way to do away with that annoying 'change this' line. (**AFilename requestFilename:** line is just as bad.)
>
> There should be a method something like
>
> ```
> baseDir := Filename whatIAmBeingFiledInFrom.
> ```
>
> Something like the C and Shell script standard of setting **ARG[0]** to the program's filename, but for **fileins**.
>
> Does such a method exist somewhere that I am unaware of?
>
> **ifFalse:**[ParcPlace please consider this an enhancement request].

I'm afraid this is also one of those questions without an easy answer, but Jan Steinman (steinman@is.morgan.com) has some good starting points:

A neat hack that I added to Tek Smalltalk some years ago was to give the **fileIn** a receiver, which is quite easy to define as the **Stream** being filed in from. Then, it becomes a simple matter of sending messages to 'self' in the **fileIn**, such as:

```
(self directory oldFileNamed: 'nextFile') fileIn!
```

(That's an old Tek Smalltalk idiom—kids, don't try this at home!) I had used this to provide a dependency mechanism, whereby a **fileIn** could determine if what it needed was present, and if not, it could go load it!

Now I'm using Envy, and therefore have no need of such things, and have not tried to do them in PPS Smalltalk. As a start, look at **PeekableStream>>fileIn** and try changing:

```
Object evaluatorClass
  evaluate: self nextChunk logged: ...
```

to:

```
Object evaluatorClass
  evaluate: self nextChunk for: self logged: ...
```

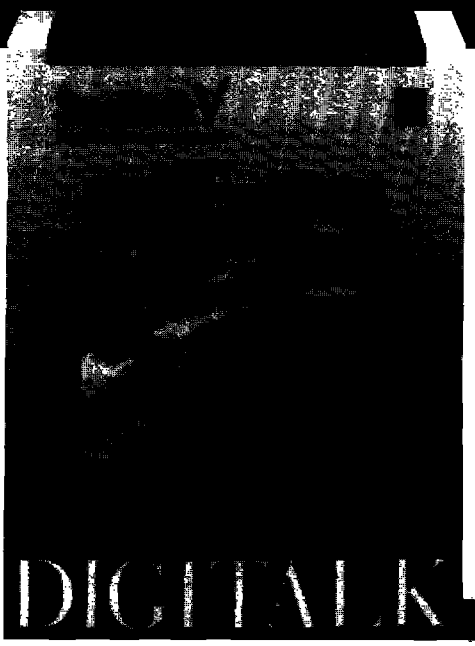
This will cause 'self' in the fileIn to refer to the Stream being read. Then you can do things like:

```
| baseDir |
baseDir := FileDirectory fullPathFor:
  self ioConnection name!
```

in your **fileIn** code. Be careful of 'self' if the **fileIn** code might not be a file, since 'self' could be an instance of **PeekableStream** (which has no **ioConnection**), or **ioConnection** might be an instance of **ExternalConnection** (which has no name). Disclaimer: I have not done any of this in PPS Smalltalk! Browse the **Stream** classes and **FileConnection** to discover other neat things you might do with this mode. Happy hacking! ■

Alan Knight is a researcher in the Department of Mechanical and Aerospace Engineering at Carleton University, Ottawa, Canada, K1S 5B6. He currently works in ParcPlace Smalltalk on problems relating to finite element analysis, and has worked in most Smalltalk dialects at one time or another. He can be reached at +1 613 788 2600 x5783, or by e-mail at knight@mrco.carleton.ca.

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



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

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

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

ONE-STOP SHOPPING.

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

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

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

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

IMMEDIATE RESULTS.

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

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

100% PURE OBJECT TRAINING.

DIGITALK