THE Smalltalk REPORT

SIGS PUBLICATIONS

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

Table of Contents

July–August 1995 Vol 4 No 9

Features

Columns



Departments

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

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

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

Editors' Corner

John Pugh

Paul White

BEFORE WE TALK ABOUT THE TOPIC on every Smalltalker's mind these days, the ParcPlace-Digitalk merger, we would like to give a warm welcome to Ralph Johnson and Bob Hinkle, our new columnists. Ralph and Bob will be taking us "Deep in the Heart of Smalltalk," a column for Smalltalk afficianados who want to learn more of the inner workings of parts of the Smalltalk system where few people dare to tread.

As most of you are surely aware by now, Digitalk and ParcPlace have merged to form a new company that, at least for now, will be known as ParcPlace-Digitalk Inc. This obviously represents a major shift of power in the Smalltalk market. Although it certainly doesn't match the world significance of many of the major takeovers and mergers that have been occurring in the software industry such as the IBM/Lotus buyout, it does leave those of us in the software development trenches scratching our collective heads.

Probably the most commonly asked question has been "will there continue to be two separate Smalltalks?"

Although it came as quite a surprise to many (including us), it would be hard to not admit that the writing was on the wall. There were telltale signs such as Digitalk not joining the new Smalltalk Industry Council (STIC) and deciding not to have their developers conference this year. And there certainly were many rumours that Digitalk was a takeover target, although names such as Microsoft and HP were far more commonly referenced as suitors than ParcPlace.

This merger leaves some obvious questions. Similar to the IBM/Lotus deal, Digitalk and ParcPlace appear to have quite different corporate cultures, with different philosophies with respect to engineering, customer support, product development, and marketing. Moreover, their two products represent quite different implementations of the Smalltalk language. Although their base class libraries are very similar, most of the facilities that allow Smalltalk to communicate with the outside world are implemented in quite different ways. The user interface architectures, for example, are quite different.

Probably the most commonly asked question has been "will there continue to be two separate Smalltalks?" But no matter what the answer is to this question, many other questions remain. If there are two separate products, how will they differentiate between them? Will one become the "low-end" product and the other a more deluxe version? Or will one product be sold as providing tight host integration where the other stresses cross-platform portability? If they do decide there will be only one product, will it be one of the existing ones, or something brand new? And where does that leave their collective installed base of customers?

Beyond the language and platform facilities there are other issues. Will Digitalk's Team/V team programming tool (which ironically was originally designed to run in the ParcPlace environment) become the standard for both Smalltalks, or will ENVY remain the configuration management tool of choice for ParcPlace and many Digitalk users? This is a significant issue for existing clients, because switching from one facility to the other is no simple task. Similarly, what are the implications for third-party products such as WindowBuilder, Repertoire, Object Explorer, etc? Both companies have separately expressed a genuine desire to bring VARs on board to support their products; where does this merger leave them?

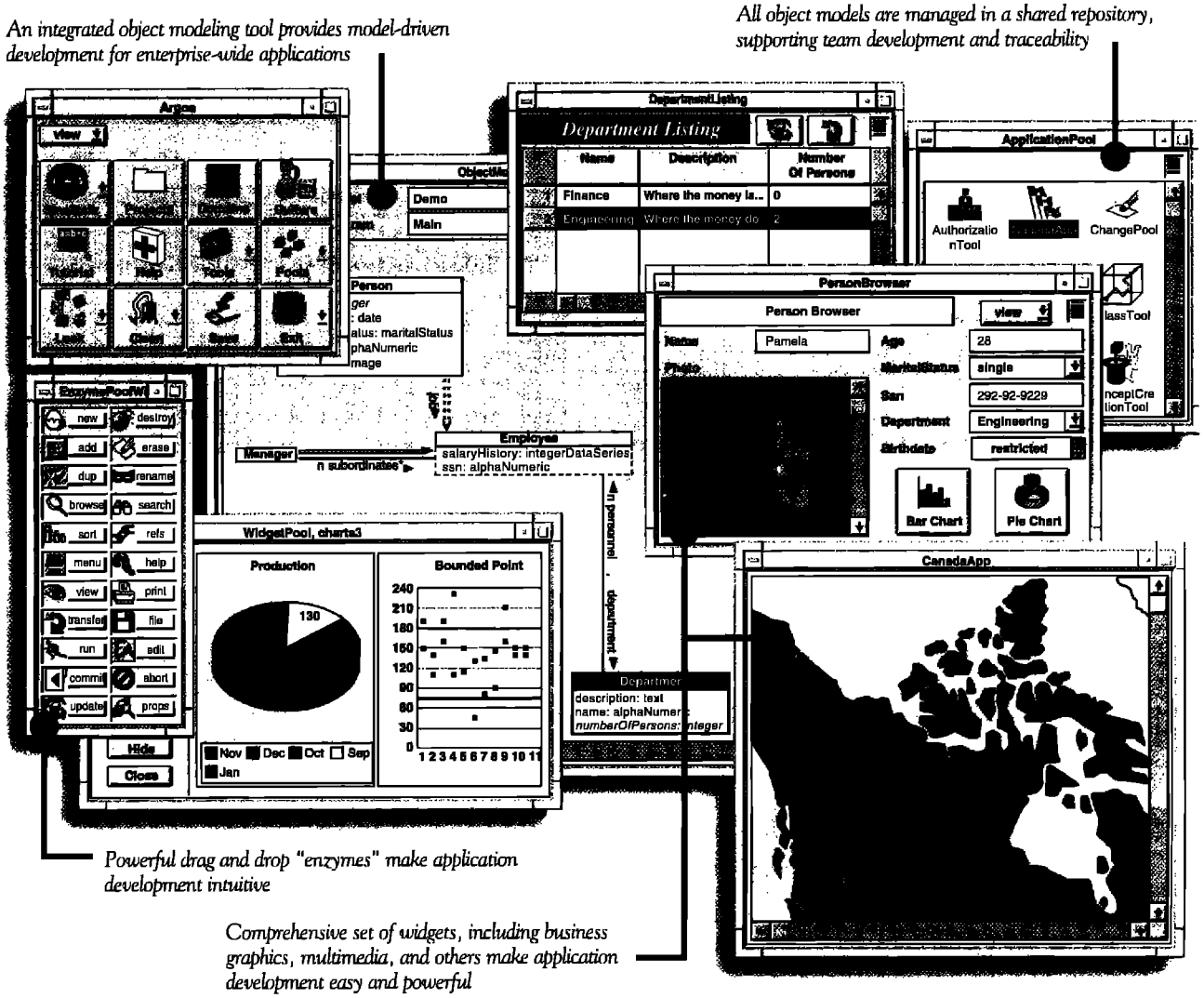
Having listed so many questions to be answered, we should be quick to point out that there are some obvious positives to the merger. The most tangible benefit is the bringing together of some of the very best minds in the programming language business, let alone the Smalltalk world. The engineering talents of both companies has never been in doubt, and if they can generate synergy from this merger, the opportunities are immense. Each company has solutions for difficult parts of the client/server application development puzzle that the other lacks. Together, they have the opportunity to build a product line that is an even more formidable competitor in the client/server arena.

The new company has promised to present their vision for the future at their upcoming (joint!) Developers Conference in San Jose at the end of July. I'm sure it will prove to be a lively event, with many anxious corporate clients wanting to know the future of their development tool. Only time will tell whether this move opens up new opportunities for the Smalltalk community. We certainly wish this new endeavour well, and wish good luck to our many friends at both these organizations.

Enjoy the issue!

Introducing Argos

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



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

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

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

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

VERSANT
The Database For Objects ™

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

Remembrance of things past:
Layered architectures for Smalltalk applications

Kyle Brown

DIDN'T YOU ALWAYS HATE IT when your father started off a sentence with "in my day...?" I know that it always irritated me, and as an adolescent I swore that I would never succumb to the temptation to start a sentence that way myself. Well, teenage oaths not withstanding, I'm going to do it anyway.

When I first learned Smalltalk, a little over six years ago, I was taught by my mentor Sam Adams that Smalltalk applications are built in layers (see Fig. 1). Having come from an engineering background where I was strongly influenced by the layered architecture of the OSI seven-layer communication model, this seemed only fitting and proper. Layered architectures promote good software design by separating concerns of one layer from another, reducing the complexity of the application of a whole, and encourage reuse both within the elements of a layer, and between layers (for a discussion of layered architectures in computer networks, see Tannenbaum[1]).

SMALLTALK APPLICATION LAYERS

The four layers that I was taught comprise a good Smalltalk application are:

1. **The GUI layer**. This is the layer where the physical window and widget objects live. Any new user interface widgets developed for this application (an activity that seems to have been more common several years ago than today) would also be put in this layer. In almost all cases today, this layer is completely generated by a window-builder tool.
2. **The Mediator layer**. This layer is partially generated by the window-builder and partially coded by the developer. The primary classes of this layer have been variously called "ViewManagers" or "ApplicationCoordinators" in Digitalk's Smalltalk products, and "ApplicationModels" in Parcplace's products. This layer mediates between the various user interface components on a GUI screen and translates the messages that they understand into messages understood by the objects in the domain model.
3. **The domain model layer**. This is where the real "meat" of the application resides. An object-oriented analysis and design should result in a set of classes that primarily reside in this layer. Examples of the type of objects in this layer would include Orders, Employees, Sensors, or whatever is appropriate to your problem domain.
4. **The infrastructure layer**. This is where the objects that represent connections to entities outside the object world reside. Examples of objects from this layer include SQLTables, 3270Terminals, SerialPorts, and the like.

Now that you've seen the description of these architectural layers, you might be saying to yourself, "Well, of course; I always build my applications like that." If so, the rest of this article might seem familiar to you. On the other hand, if you're scratching your head and going "But why..." or, even worse, smacking your forehead and declaring "Why didn't I see that!" then please read on. I assure you your projects will benefit.

Something that stood out to me during the recent Smalltalk Solutions conference is how rarely novice Smalltalk programmers see their applications as divided into layers. I believe that this is a result of the wonderful new tools that the Smalltalk vendors have provided to us over the past few years.

Back in the old days, Smalltalk programmers would lovingly handcraft each of their classes one at a time. You don't really gain an appreciation of how difficult it is to build a new graphics class or a database framework until you've coded one yourself. However, in the modern point-and-click world, building a complex order-entry screen seems as simple as drawing the GUI using your window-builder of choice, telling it to generate the resulting Smalltalk classes, then hooking those classes into the database by using the vendor-supplied database connectivity classes.

While the new tools have vastly increased our productivity as programmers, it has become easy to lose sight of the bigger picture. While it is possible to completely code an application without ever creating a new class of your own, it is not necessarily desirable. Part of the appeal of object-oriented programming is the ability to create reusable classes that represent your problem domain. In that way, a month or a year from now when you (or someone else in your organization) is getting ready to work on the next application, you can pull those classes off the shelf and use them as is, or specialize them through subclassing. It is this reusability that gives O-O programmers the productivity advantage over their 3GL programmer peers.

Let's say you've just coded an order-entry screen the way I described above (see Fig. 2). Six months later your customer comes up to you and says "I've decided that I

don't like this GUI you've built; I want a drag-and-drop interface. Oh, and by the way, we're changing from a relational database to an ODBMS next week. Can I have the changes by Friday?" At this point you might be tearing out your hair and considering looking into a promising new career in food service. Or, if you had instead coded your application into layers, you could be taking it all in stride, saying "OK, first I'll have to redraw the GUI; that'll only take a little while; then I can start thinking about the rest of the problem."

RULES FOR PROPER LAYERING

So, now that I've convinced you that layering your application is a good idea, how do you actually go about it? At KSC we believe that applications are like sandwiches. We follow the approach of building a Smalltalk application inside out; we begin with the hamburger and work our way out to the bun.

The primary focus of your effort in a non-trivial Smalltalk application should be in defining what domain objects you have, then building and testing them. We have found that if you begin with a statement of what your problem is, and then develop an O-O model using a behavioral OOA&D methodology, you can usually get a good handle on the domain layer of your system. For instance, if you are building an order management and

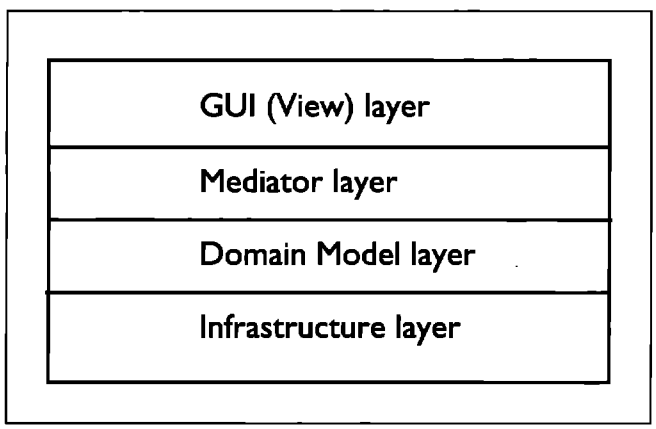


Figure 1. Smalltalk application layers.

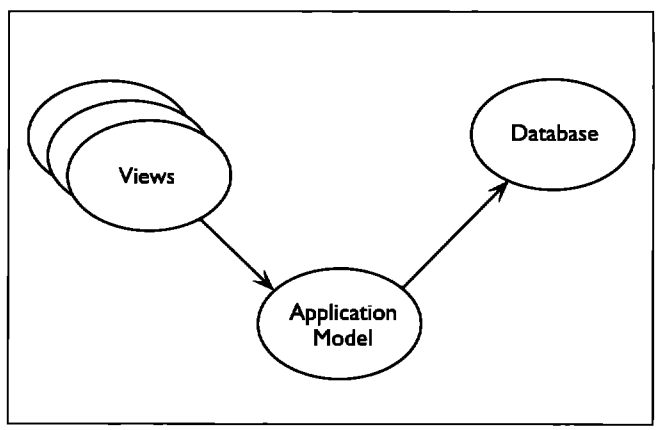


Figure 2. Missing domain model.

inventory system, you might discover objects like Order, Warehouse, and Good and investigate how they interact with each other. You can then proceed from a design model to building a prototype of these objects. The key here is to discover how the objects interact, and how you can take advantage of the paradigm to make your objects more understandable and reusable. The domain layer should be developed, as much as possible, without undue consideration as to how the database will be implemented, or how the GUIs will look. While these considerations are important to the application as a whole, they should not be allowed to "pollute" the purity of your object model.

After you have built and tested your prototype object model, you can then begin to work your way out to the layers surrounding the domain model in Figure 1. Let's begin by looking at the development of the ApplicationModel layer. As I stated earlier, this layer should be primarily for mediating between the different elements on your GUI, and translating messages from the GUI into messages understood by your domain model. It should follow the intents of the Mediator pattern and the Adapter pattern from Gamma.[2]

In a nutshell, a Mediator is an object that "encapsulates how a set of objects interact."[2, p.273] An example of this kind of mediation is that an ApplicationModel may disable a set of buttons or menu items based upon the state of other buttons or menu items. The ApplicationModel keeps these widgets from knowing about each other, and promotes good factoring of the design. It doesn't make much sense to involve the domain layer in these, purely user-interface actions, so the Mediator also insulates the two from each other.

An Adapter "converts the interface of a class into another interface clients expect."[2, p.139] GUI Widgets have one interface that they respond to—they are concerned with the state of their selection, what they are displaying, etc. On the other hand, the domain model is concerned with a different sort of interface—it is concerned with the state of Orders, or how Employees are related, etc. An ApplicationModel should adapt the one interface to the other. It should not attempt to take over the responsibilities of either, but make their communication smooth.

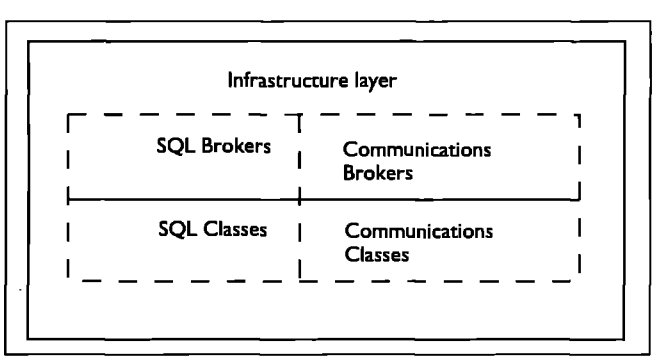


Figure 3. Infrastructure microlayering.

In general, your ApplicationModels should be thin, dumb, and small. By these characterizations I mean that an ApplicationModel should have few methods (thin vs. fat), the methods themselves should serve only to translate and mediate, rather than control (dumb vs. smart), and that the ApplicationModel objects themselves should have little state, or few instance variables (small vs. large). Remember that the task of a GUI should be to present a face to the world that reflects the state of the objects in your application. Resist the temptation to represent the state of the application in the GUI itself.

A common example of cooperation between the Mediator layer and domain model objects would be in the validation of values entered into a GUI. Many novice designers will immediately code all validation logic into the mediator layer, without considering the consequences that decision makes. If the information that you want to validate is held in an object in the domain model, does it really make sense for the range checks and other validations to be made in different object in another layer? The principle of encapsulation indicates that the behavior should accompany the information. On the other hand, the GUI must represent the visual aspects of the validation; this would include warning the user if an incorrect value is entered or preventing the value from being accepted. The two objects should cooperate along a well-defined path of communication rather than stuffing the responsibility all in one object or the other.

Just remember to consider the implications of each design decision to determine if it violates proper layering...

Another example of improper distribution of responsibility between the Mediator layer and the domain model is in the representation of calculations, or intermediate results in a calculation. If your application calls for you to display the line items in an order, and the sum of their costs, should the calculation of that sum reside in the ApplicationModel class, or in the Order that contains the line items? While you can code the calculation into the ApplicationModel, it would be more reasonable to allow the Order itself to do this calculation. In that way, the summed cost would be available to other presentation options; such as a paper report, or a summary report of many orders. Following these rules leads to objects with a better distribution of responsibilities, which results in more reusable objects.*

Finally, let's take a look at the division on the other end; between the domain model and the infrastructure of your application. I think it's become abundantly clear by now that it is an absolute no-no for the Mediator layer

* Wirfs-Brock[3] contains a good discussion of the benefits of well-distributed behavior.

to communicate directly with the infrastructure layer, but what is appropriate communication between the domain and infrastructure layers? In general terms, the communication should preserve encapsulation boundaries. A "microlayering" approach might work out best (Fig. 3). Just as novices are tempted to put all their logic in their Mediator code, an equally naive approach is to put all the database knowledge into the domain classes themselves, or (slightly better) in their metaclasses.

A better approach to building connections between domain and infrastructure layers is to build a layer of "helper" or "broker"objects.† A "broker"‡ is an object that serves as an adapter between the domain object that must communicate to the outside world, and the communication medium, be it a network protocol, a mail protocol like SMTP, or a relational database. Again, the primary advantage here is the preservation of encapsulation; if you can encapsulate the knowledge of a protocol or interface into one set of (reusable) objects, and provide an adapter between them and the domain layer, you will be better able to change one without necessitating changes to the other.

† For an discussion of architectures for relational database interaction, see Vasan.[4]

‡ By "broker," I'm not referring to a CORBA-style Object Request Broker (ORB). A broker is any object that adapts an object model to a non-object-oriented procedural interface. Brokers must, by necessity, know a little about each world to bridge the gap between them.

FINAL NOTES

While applying a layered architecture to your applications will not be a panacea for all your software ills, it may alleviate some of the more grevious symptoms. Just remember to consider the implications of each design decision to determine if it violates proper layering or enhances the reusability of the individual classes by supporting the architecture.

Acknowledgement

The author thanks Bobby Woolf for pointing out the proper patterns that ApplicationModels should follow, and all his peer reviewers at KSC for their help and advice.

References

1. Tannenbaum, A. COMPUTER NETWORKS, Prentice Hall, Englewood Cliffs, NJ, 1988, pp. 9–14.
2. Gamma, Helm, Johnson, and Vlissides. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.
3. Wirfs-Bock, R. Characterizing your application's control style, SMALLTALK SOLUTIONS '95 CONFERENCE NOTES, New York, 1995.
4. Vasan, R. Techniques for object and relational integration, OBJECT MAGAZINE 3(1):52–53, 60, 1993.

Kyle Brown is a Senior Member of Technical Staff at Knowledge Systems Corp. He has been developing industrial applications in Smalltalk for over five years. As part of his consulting practice, he has built applications for Engineering, MIS, and scientific computing. Since joining KSC, Kyle has enjoyed teaching the principles of reuse and good O-O design to a variety of clients through the KSC Smalltalk Apprentice Program. He can be reached via email at kbrown@ksccary.com.

Segregating application and domain

Tim Howard

THIS IS THE THIRD ARTICLE IN A SERIES of three dedicated to the topic of segregating application information and domain information in VisualWorks application development. The first article presented the case of why it is essential that an application have a strict segregation between its application information and its domain information. The second article covered the implementation of domain objects, which are the keepers of the domain information. This third article discusses the application classes, which provide the user interface for the domain objects.

This article introduces what I refer to as the *domain adaptor architecture,* which is a framework for building applications in VisualWorks. This archictecture is a specialization of application model architecture and is designed to work specifically with domain objects. First I will introduce the domain adaptor, which is a special type of application model developed for viewing and editing a domain object. Then I will discuss how domain adaptors bind the user interface to the information in the domain object. The full source code for the domain adaptor architecture, along with examples, is available from the archives at the University of Illinois (st.cs.uiuc.edu).

DOMAIN ADAPTOR

In the second article of this series, I talked about domain objects, the keepers of the domain information. These domain objects do not exhibit model behavior nor do they know how to present themselves in a user interface. Now what we need is a special type of application model architecture that allows us to easily build applications for displaying and editing these domain objects. I refer to this type of architecture as the *domain adaptor architecture* because it adapts purely domain information to a user interface. At the center of the domain adaptor architecture is the domain adaptor. A *domain adaptor* is a type of application model that provides a user interface for a domain object and allows the user to view and edit that domain object.

Each class of domain adaptor is designed for a specific class of domain object. For example, suppose we have the domain class EmployeeReview, which describes all the information for a single employee review. To view an instance of such a class (a domain object) we might create a type of domain adaptor called EmployeeReviewUI. There can be more than one domain adaptor class for each class of domain object, but each class of domain adaptor is designed specifically for only one type of domain object. As an example, we can design EmployeeReviewUI, ShortFormEmployeeReviewUI, and LongFormEmployeeReviewUI to operate on an EmployeeReview object—each presenting the employee review information in a different way. Each domain adaptor operates on only one domain object at a time. Continuing with our example, an instance of EmployeeReviewUI will present to the user one instance of EmployeeReview for viewing and/or editing.

The domain object on which a domain adaptor operates is accessed and replaced using value model protocol—the messages value and value:. In our example, when the user is done filling out the employee review in the user interface, thereby populating the EmployeeReview domain object, we can access that domain object by sending value to the EmployeeReviewUI domain adaptor. We can then place this domain object in a database, add it to a collection of such reviews, etc. When the user is ready to read or edit the next employee review, we simply send value: anEmployeeReview to the EmployeeReviewUI domain adaptor. The domain adaptor then automatically populates its user interface with the information in the new domain object.

Because each type of domain adaptor is designed for a specific type of domain object, it knows how to create a new instance of that type of domain object in the event that one is not provided at the time the user interface is opened. In the employee review example, the code:

```
EmployeeReviewUI open
```

will open a window on a new instance of EmployeeReview, because one is not initially provided. If we already had a populated instance of EmployeeReview, we would open an interface on it with the following:

```
EmployeeReviewUI openOn: anEmployeeReview.
```

The abstract implementation for all domain adaptors is defined in DomainAdaptor:. DomainAdaptor is a subclass of ExternalApplicationModel and defines two instance variables: domainChannel and domainIsChanging.

Because the superclass ExtendedApplicationModel[1] is a subclass of ApplicationModel, all domain adaptors are application models and, therefore, define and operate user interfaces. Furthermore, all domain adaptor classes inherit the development features defined in ExtendedApplicationModel. The domainChannel instance variable is a ValueHolder that references the current

domain object. The domainIsChanging instance variable is a Boolean that is true when a change of domain is in progress. The DomainAdaptor class implementation can be divided into three parts: domain object management, aspect support protocol, and interface opening protocol. Domain object management involves managing the domain object itself. The aspect support protocol is a set of methods that allow you to easily set up aspect models that operate on the information in the domain object. The interface opening protocol is an extension of the interface opening protocol defined in ExtendedApplicationModel, which allows a domain adaptor to be opened and initialized to an existing domain object in a variety of ways.

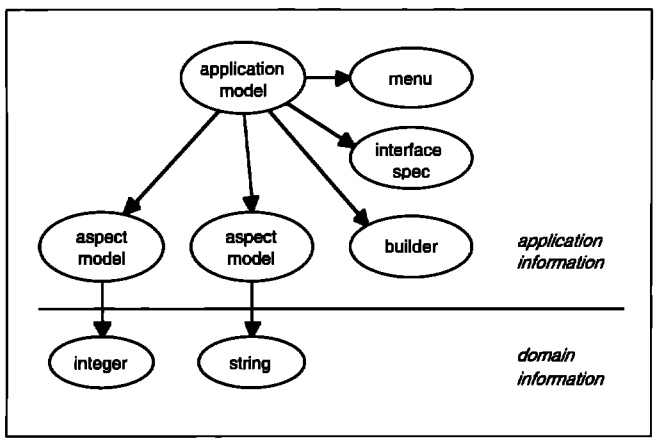


Figure 1. Application model object diagram.

In the first article of this series, I presented an object diagram of an application model. For convenience, this diagram is provided again in Figure 1. Figure 2 is the object diagram for a domain adaptor. Notice that in Figure 1 the domain information is logically related via the application model. In Figure 2, however, the doman information is logically related via the domain object. Thus the domain information can be easily uncoupled from the application information and placed in a database. Likewise, another domain object of the same type

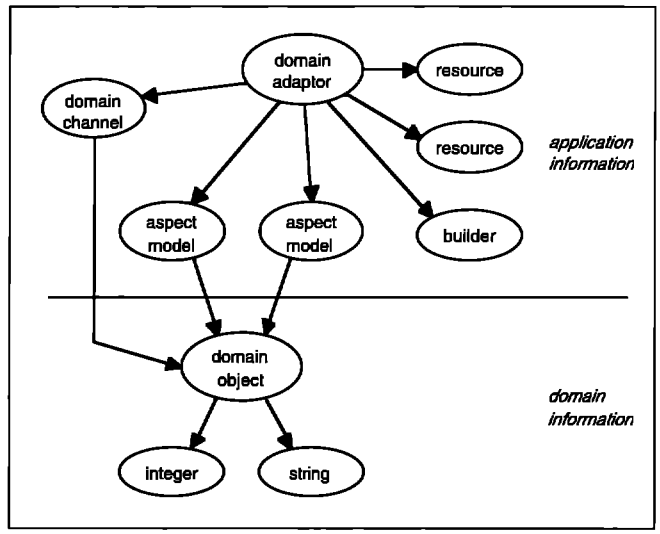


Figure 2. Domain adaptor object diagram.

can be retreived from the database and plugged back into the domain adaptor. Accessing the domain object and inserting a new domain obejct is as easy as sending the messages value and value:, respectively.

ADAPTING THE DOMAIN INFORMATION

In designing a domain adaptor, our main goal is to define aspect models that operate on the various aspects of information in the domain object. When the various interface components bind with these aspect models, the result is a user interface that views and edits the information in the domain object.

In the second article of this series, I categorized the domain information into atomic objects (such as numbers, strings, and dates), collections, and other domain objects. Consider an Applicant domain object used to describe someone applying for a job and having the following instance variables:

Variable	Type
name	String
ssn	String
references	SortedCollection of Strings
address	Address

The Applicant domain object references objects of each of the three categories of domain information. The name and ssn instance variables are atomic in nature. The references instance variable is a collection. The address instance variable references yet another domain object, an Address object, whose instance variables are defined as follows.

Variable	Type
street	String
city	String
state	String
zip	String

An object diagram for the Applicant domain object is shown in Figure 3.

To provide a user interface for viewing and editing an Applicant object, we need a special type of domain adaptor; therefore, we define the class ApplicantUI as a subclass of DomainAdaptor. Each class of domain adaptor needs to implement the domainClass instance method, which indicates the type of domain object for which the domain adaptor is designed. In our example, we would define the following in the protocol "domain accessing":

```
domainClass
    ^Applicant
```

We can also draw the user interface as is shown in Figure 4. The address portion of the user interface is not explicitly drawn in this canvas but is actually a subcanvas, as will be demonstrated shortly.

Now we need to define aspect methods for our domain adaptor which bind the information in the domain object to the various components in the user interface. First we will consider the atomic objects—strings, dates, integers, floats, and Booleans. Such information is usually presented to the user using input fields, text editors, check boxes, and radio buttons. VisualWorks already provides a mechanism, the AspectAdaptor, by which we can adapt a domain object's atomic information such that it can be displayed by these interface components. An AspectAdaptor is a value model whose value actually belongs to some other object—in our case, the domain object. When several AspectAdaptor objects operate on the same domain object, it is convenient to keep that doman object in a ValueHolder. Fortunately, each domain adaptor has such a ValueHolder, its domainChannel instance variable. Because setting up an AspectAdaptor can be somewhat complicated, the DomainAdaptor class defines certain aspect support methods to set up the AspectAdaptor for us. In the applicant example, we need an AspectAdaptor for both the name and ssn attributes of the domain object. Therefore, in the

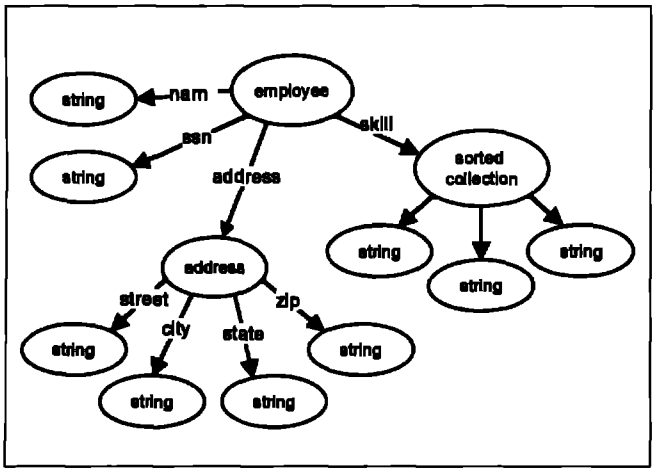


Figure 3. Applicant object diagram.

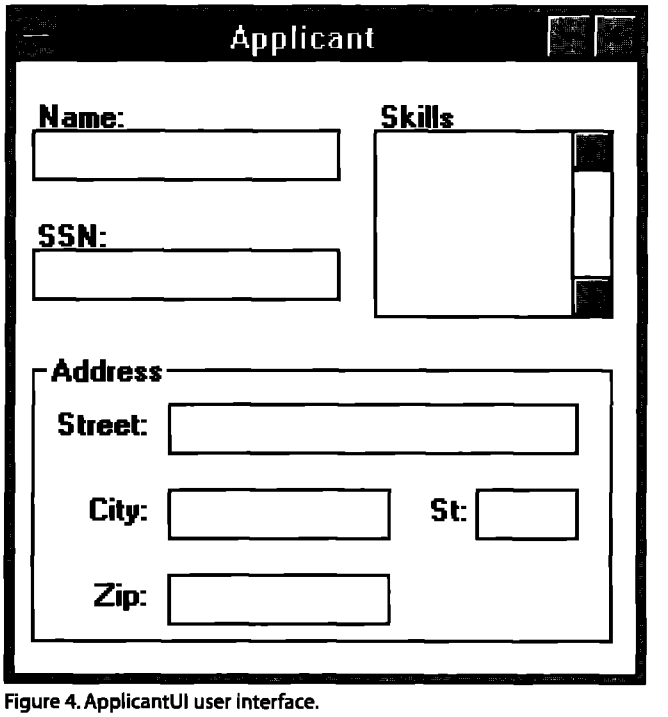


Figure 4. ApplicantUI user interface.

"aspects" protocol of the ApplicationUI class we define the following two methods:

```
name
    ^self aspectAdaptorFor: #name

ssn
    ^self aspectAdaptorFor: #ssn
```

Notice how simple these methods are. The method aspectAdaptorFor: aSymbol is defined in DomainAdaptor and it automatically sets up an AspectAdaptor for the domain object instance variable named by aSymbol. The domain adaptor's domainChannel instance variable is used as the subject channel for the AspectAdaptor so that whenever the domain object in domainChannel is replaced with another, the AspectAdaptor is automatically switched over to the new domain object, and, furthermore, the corresponding interface component is updated with the new information. A useful variation of the aspectAdaptorFor: method is the aspectAdaptorFor:changeMessage: method, which will set up the AspectAdaptor such that whenever its value is changed, a change message is dispatched to the application model.

The AspectAdaptor works very well for the atomic type information, but what about the collections contained by our domain objects? Collections are typically presented to the user in list components. For example, we want to display the applicant's skills in a list component and also allow the user to add and remove skills. What we need is a collection version of the AspectAdaptor. We need something that will allow a domain adaptor not only to display a domain object's collection in a list component, but also permit the user to add and remove elements from that collection. Furthermore, when the current domain object is replaced, this new type of adaptor must switch its focus to the collection in the new domain object and have the list component redraw itself with the new information. For this purpose, I have created the CollectionAdaptor class. DomainAdaptor defines aspect support protocol for setting up a CollectionAdaptor. For example, to have our ApplicantUI show the skills of an applicant, we would add the following method to "aspects" protocol of ApplicantUI:

```
skills

    ^self
        collectionAdaptorFor:   #skills
        collection:  #skills
```

We can now easily adapt the atomic objects and collections in our domain objects to the user interface managed by the domain adaptor. But what about domain objects that contain other domain objects? In our example, our Applicant object holds on to an Address object—which is itself a domain object. Do we have an adaptor for it? Yes, its called a domain adaptor! What kind of interface component do we use to display an Address object? A subcanvas managed by an AddressUI domain adaptor! The domain adaptor architecture is fully recursive. Furthermore, all

three types of adaptors operate on value model protocol. The messages value and value: access the domain information from the adaptor whether that information is atomic, a collection, or another domain object. In the applicant example, we would add the following method to the "aspects" protocol of ApplicantUI:

```
address
    ^self domainAdaptorFor: #address model: AddressUI
```

This method automatically instantiates a domain adaptor of the type AddressUI. This AddressUI domain adaptor will operate on the Address object contained by our Applicant object and display that Address object in a subcanvas. AddressUI is just another domain adaptor originally

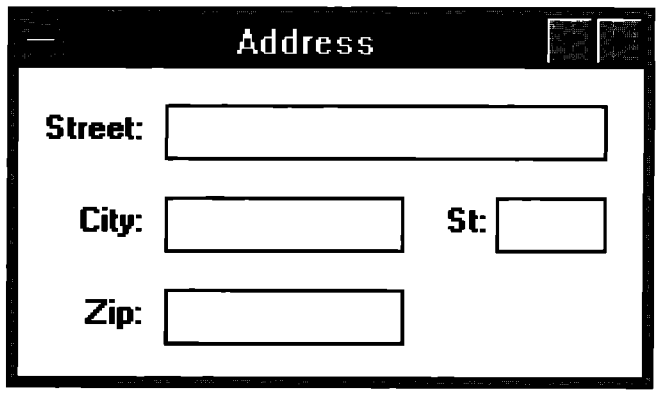


Figure 5. AddressUI user interface.

designed as a window interface for an Address object, as shown in Figure 5.

It is important to emphasize that we are using another domain adaptor, developed completely independently of ApplicantUI and Applicant objects, but that can be easily incorporated into the ApplicantUI domain adaptor. This allows us to take either a bottom-up or top-down approach to building user interfaces for our domain objects. It also facilitates reuse because an AddressUI can be used independently or incorporated into several other applications requiring an address. As domain objects become aggregations of other domain objects, their correspdonding domain adaptors are just mirror aggregations of other domain adaptors. In this way, domain adaptors can be designed for even the most complicated domain objects.

I would like to point out that all the aspect support protocol in DomainAdaptor is defined in such a way that our aspect models do not need corresponding instance variables defined in the domain adaptor class. This provides for a much cleaner class definition (see "Extending the Application Model"). For a complete list of the aspect support protocol and how it is used, browse the class DomainAdaptor.

Table 1 summarizes the types of domain information, the corresponding interface component used to display that information, and the adaptor used to connect the domain information to the interface component.

Table 1. Adapting interface components to domain information.

Domain Information Type	Interface Component	Model Adaptor
String, Number, Date	Input Field	AspectAdaptor
Text	Text Editor	AspectAdaptor
Symbol	Radio Buttons	AspectAdaptor
Boolean	Check Box	AspectAdaptor
OrderedCollection SortedCollection	List	CollectionAdaptor
DomainObject	Subcanvas	DomainAdaptor
DomainObject	Window	DomainAdaptor

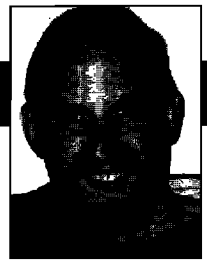
SUMMARY

This article introduced the domain adaptor architecture, which is a framework for building VisualWorks applications based on domain objects and a strict segregation of application and domain information. The centerpiece of the domain adaptor architecture is the domain adaptor—an appliction model that knows how to operate on a domain object. The domain adaptor keeps its domain object in a ValuHolder referred to as the domain channel. Each time the domain object is replaced by a new domain object, the domain adaptor updates its interface with the new information. A domain adaptor uses model adaptors to bind information in its domain object to the interface components. An AspectAdaptor binds simple information to input fields, check boxes, and radio buttons. A CollectionAdaptor binds collections to list components. Domain adaptors are themselves model adaptors and bind contained domain objects to subcanvases. There is a great deal more to the domain adaptor architecture than can be presented in a single article. Not covered are such topics as the role of dialogs, the interface opening protocol, child windows, buffered adaptors, and strategies and guidelines. If you program in VisualWorks, I stongly encourage you to obtain the source code and work the examples. After only a few hours of exploration, the merits of this approach to VisualWorks application development will be quite evident. The full source code for the domain adaptor architecture framework and examples are available from the archives at the University of Illinois (st.cs.uiuc.edu).

Reference

1. Howard, T. and B. Kohl. Extending the application model, THE SMALLTALK REPORT 3(7):1, 4–7, 1994.

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



Bob Hinkle Ralph E. Johnson

ParameterizedCompiler: A case study in making code reusable

SOMETIMES YOU CAN TELL at the start of a project that your code must be reusable. But more often, you don't realize that something needs to be reusable until you try to reuse it, and you end up trying to add reusability to existing code. This is not easy, because reusability is a result of the design of a system, not just the result of some coding tricks. So, making code reusable often requires changing its design significantly.

Recently we needed to reuse the VisualWorks compiler to implement a new breakpoint mechanism. Unfortunately, the compiler was not designed to be reused the way we wanted. As a result, we had to rewrite it to be more parameterizable and easier to customize.

This column exposes some of the more arcane inner workings of Smalltalk, and shows why it is both useful and powerful to allow programmers reflective access to these parts of the system. It also shows several common techniques for making Smalltalk programs more reusable. The solution described here is based on VisualWorks Version 2.0, although it can be adapted to previous versions of Smalltalk-80.

THE PROBLEM

Our previous article[1] described how to debug the behavior of individual objects using lightweight classes. To make debugging easier, and as something of a side project, we also introduced breakpoints, a common feature of modern programming environments that had so far been represented in Smalltalk-80 only by its poor cousin, the halt message. While the breakpoints we introduced had several benefits over self halt—notably their independence from the Change List and their ease of addition and removal—they also had several limitations. In particular, the breakpoints could only be set at the very beginning of a method, and they were unconditional. We want to implement breakpoints that can exist between any two statements in any block and that can be either absolute or conditional. Conditional breakpoints stop execution only if some expression evaluates to true. This expression is a general block of Smalltalk code evaluated in the context of the breakpointed method, allowing access to method and block parameters and temporaries.

These new requirements force changes to our previous implementation. Because breakpoints can be conditional, we must be able to insert an arbitrary block of code anywhere breakpoints are allowed, and provide an interface for users to enter and edit breakpoint condition strings. Because breakpoints can occur in the middle of a method, we need a more general mechanism for installing breakpoint code. We used to wrap a new breakpoint method around the unchanged original method, but now we must insert breakpoint code in the body of the breakpointed method. Furthermore, the code we insert must not only work correctly but also respect the source code map, so that stepping through breakpointed methods in the debugger works as the user expects. Finally, because breakpoints can occur within a method, their position must be indicated graphically in a method's source. Combined with our continued desire to insert breakpoints without affecting the Change List, this implies the need for two levels of source code for breakpointed methods: one to display in browsers, and the other to store to and retrieve from disk.

While the current compiler, comprising Compiler, Parser, and their co-workers, is very powerful, we have to extend it significantly to overcome these various difficulties. We need to change the compilation process, so we can insert our breakpoint code into the method's body and produce methods with distinct display and stored source texts. Extending an existing component this way is a typical step in system building, and many of the techniques for implementing design and redesign are equally typical, falling into previously identified patterns (see Gamma[2] for a catalog of common patterns in the object-oriented world). We'll apply patterns called Factory-Method, Strategy, and Visitor, as well as a program refactoring technique we call Trail Splitting.

Bob Hinkle is an independent Smalltalk consultant and writer. His current focus is the improvement of existing tools and the creation of new tools to revitalize the Smalltalk environment. He can be reached at hinkle@primenet.com. Ralph Johnson learned Smalltalk from the Blue Book in 1984. He wrote his first Smalltalk program in the fall of 1985 when he taught his first course on object-oriented programming and design. He has been a fan of Smalltalk ever since. He is the only author of DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE to regularly program in Smalltalk, and continues to teach courses on object-oriented programming and design at the University of Illinois.

THE SMALLTALK COMPILER

Because most of the changes in this project center around the process of compiling code to produce new methods, we will examine how the compilation process currently works to see how it can be extended. The current process involves six major steps:

1. The user selects accept in the TextView of a Browser.
2. The Browser passes its current text to the currently selected class using the message #compile:classified:, which forwards to ClassDescription>>compile:classified:notifying:.
3. The class creates a new Compiler—the new compiler's class is actually defined in the class method #compilerClass, but this always returns Compiler—and passes the source text using the message #compile:in:notifying:ifFail:.
4. The Compiler initializes itself and eventually calls the method that does the work, #translate:noPattern:ifFail:needSourceMap:handler:.
   (a) The Compiler creates a Parser, which it uses to create a parse tree from the source text. The Parser works with a ProgramNodeBuilder to create this tree, which consists of instances of the various subclasses of ProgramNode.
   (b) The compiler creates a new CodeStream and tells the root of the parse tree, using the #emitEffect: message, to generate byte codes into the CodeStream. The name scope for variables in the parse tree is resolved using a subinstance of NameScope created by the compiler.
   (c) The resulting CompiledMethod is packaged into a MethodNodeHolder, which is returned as the result.
5. The Class obtains a CompiledMethod from the MethodNodeHolder using the message #generate (which is currently trivial, returning the CompiledMethod from step 4(c)).
6. The Class updates the Change Set, writes the source text out to the Change List, sets the CompiledMethod's sourcePointer, and adds the new method to its MethodDictionary.

Most of the work is done in step 4. Each substep of 4 is a major production point, a place where an important object is produced and returned to be used in the next step. Step 4(a) creates the parse tree, step 4(b) creates the byte code stream, and step 4(c) creates the method itself. Nine classes of objects are instantiated in steps 4(a) through 4(c) (counting parse tree nodes as subinstances of ProgramNode), and there is much built-in flexibility in the process. The compiler's and the parser's classes are specified in #compilerClass and #preferredParserClass, respectively. Furthermore, new methods and blocks are instantiated in CodeStream using the messages #methodClass and #blockClass sent to the compiler. In addition, instances of CodeStream, NameScope, and MethodNodeHolder are created by Compiler's messages #newCodeStream, #scopeForClass:, and #newMethodHolder. These implementations are examples of a design pattern called FactoryMethod, which works by using methods either to specify a class for instantiation or to produce a new instance. Either way, Factory Methods can be overridden in subclasses, making it relatively easy to introduce new kinds of collaborators into a complex process.

While quite flexible already, this process has some limitations. ProgramNodeBuilder is hard-coded into #translate:noPattern:ifFail:needSourceMap:handler:, and ProgramNodeBuilders always produce subinstances of ProgramNode. More importantly, while Factory Methods are helpful, they do require subclassing to introduce new object types into the compilation process. We will make this process more flexible, so that programmers can define new behavior at each production point, by taking advantage of the fact that classes are objects. The various classes used in the compilation process will be stored in instance variables of a new compiler, so that new kinds of collaborators can be easily introduced by setting these variables to new values. Finally, if we think of compilation as a production line, there are four stages intermingled with the three production points of step 4. At each stage, we may wish to transform the object(s) flowing through the production line to affect compilation. In the first stage we can transform the input text, in the second the parse tree, in the third the bytecode stream, and in the fourth the compiled method itself. Our breakpoint project exploits two of the production points and one of the intervening stages, and in future articles we will see uses for specializing the other production points and transformation stages.

It is both useful and powerful to allow programmers reflective access to {the inner workings} of the system.

REFACTORING THE SMALLTALK COMPILER

Our new compiler, ParameterizedCompiler, allows programmers to override behavior at each production point and transformation stage, providing great flexibility in the compilation process. It achieves this flexibility by parameterizing each class of objects used during compilation, allowing any subset of these classes to be replaced with new specializations. Also, the ParameterizedCompiler lets you intervene at each transformation stage by sending messages to its client. To support this flexibility, the ParameterizedCompiler requires a new collaborator, who supplies it with the classes it should use and responds (even if trivially) to the transformation-stage messages. This collaborator will be an instance of the new class MethodProducer.

While these tasks could be performed by ParameterizedCompiler itself, there are several reasons why it's better to introduce a new object instead. The Compiler in Smalltalk, as the entry point to the entire compilation aggregate, is already a big, complex object, and adding new

responsibilities only makes it bigger and more complex. This makes it difficult to understand in its own right, and also difficult to extend with new interventions, because the programmer must determine which methods and interactions have to do with the core job of compiling and which are designed to be specialized. Specialization becomes easier when the methods intended to be overridden are concentrated within the locus of a single object. This is an application of the Strategy pattern, which bundles algorithms in different objects, allowing them to be varied independently from their clients. Thus, different kinds of Behaviors can exploit this flexibility to create their own compilation algorithms, by simply defining an extension of MethodProducer that introduces new kinds of collaborators or new responses to transformation-stage messages. For example, the lightweight classes of our previous article[1] only need to introduce a new kind of method that stores its source text locally. Supporting breakpoints requires an altered parser and a new method class, as well as manipulation of the parse tree in the second transformation stage. In a future project to add active variables, we'll use a MethodProducer with a specialized parser, a new ProgramNodeBuilder that provides a new kind of ProgramNode, and a new extension of NameScope. Each of these examples can be handled by a slightly modified MethodProducer that, by interacting differently with ParameterizedCompiler, specializes compilation to suit each particular kind of Behavior.

The class definitions for these two new cooperative classes are as follows:

```
Object subclass: #MethodProducer
    instanceVariableNames: 'client'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Parameterized Compiler'

SmalltalkCompiler subclass: #ParameterizedCompiler
    instanceVariableNames: 'producer parserClass
builderClasscodeStreamClass nameScopeClass methodClass
blockClass holderClass '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Parameterized Compiler'
```

Every class responds to the message #methodProducer by returning the default MethodProducer for building the class' methods, just as they now define their default compiler class. This class-specific MethodProducer is responsible for creating a new compiler, and, if necessary, for initializing the instance variables that specify what classes are used during compilation and responding to messages from the compiler at each transformation stage.

MethodProducer and ParameterizedCompiler are introduced into the standard compilation process by changing the compiling messages in ClassDescription and Behavior. We redefine the method ClassDescription>>compile:classified:notifying: (mentioned in step 2 above) and add a new method to which it will forward:

```
compile: code classified: heading notifying: requestor
    ^self
        compile: code
        classified: heading
        notifying: requestor
        producer: (self methodProducerForText: code)

compile: code classified: heading notifying: requestor
producer: producer
    ^producer
        compile: code
        in: self
        classified: heading
        notifying: requestor
```

The actual MethodProducer created in the first of the two methods above is provided by implementing two additional methods in Behavior:

```
methodProducer
    ^MethodProducer new client: self

methodProducerForText: aTextOrString
    ^self methodProducer
```

The first method instantiates a MethodProducer of a type suitable for the class, and classes can override this method to use new subclasses of MethodProducer. The second method, while trivial now, lets a class use different MethodProducers depending on the method and/or source code to be compiled.

HOW TO CONTACT SIGS PUBLICATIONS

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

For customer service and to order a subscription, renew, or change the name/address of an existing subscription
In the US—
P.O. Box 5050
Brentwood, TN 37024-5050
Phone: 800.361.1279 Fax: 615.370.4845
email: subscriptions@sigs.com
In Europe—
Subscriptions Dept.
Tower House
Sovereign Park
Market Harborough
Leicestershire, LE16 9EF, UK
Phone: +44(0)1858 435 302
Fax: +44(0)1858 434 958

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

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

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

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

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

MOVING CODE TO MethodProducer

The MethodProducer responds to #compile:in:classified:notifying: with a method similar to the old ClassDescription>>compile:classified:notifying:.

```
compile: code in: aClass classified: aProtocolnotifying:
aRequestor ifFail: aBlock
    | methodNode selector method |
    sourceCode := code.
    Cursor execute showWhile: [
        methodNode := self newCompiler
            compile: code
            in: aClass
            notifying: aRequestor
            ifFail: aBlock.
        selector := methodNode selector.
        method := methodNode generate.
        self storeSource: code method: method class:aClass
selector: selector classified: aProtocol.
        aClass loadMethod: method selector:
selectorclassified: aProtocol].
    ^selector
```

The differences between this method and the old ClassDescription compilation method are the three underlined messages. First, MethodProducer uses a Factory Method, #newCompiler, to create the compiler it will work with. In default cases, this method will return a ParameterizedCompiler that uses the same group of collaborators as Compiler, so compilation will proceed much as it used to do. For more specialized requirements, such as breakpoints and lightweight classes, #newCompiler will be overridden to return a ParameterizedCompiler with new kinds of collaborators.

Unlike ClassDescription>>compile:classified:notifying:, which directly added the newly compiled source code to the Change List, MethodProducer makes a separate call to store source, which provides an easy way for future specializations to extend or override (and which will be exploited to support both lightweight class methods and breakpoint methods). MethodProducer defines the source-storing message by sending #storeSource:method:selector:classified: to the designated class, which responds by updating the Change Set and Change List. Frequently, as happens here, moving a method from one class to another requires splitting off a portion that remains in the old class, where it is easier to access local instance variables and conceptually more self-contained.

Finally, ClassDescription implements loadMethod:selector:classified: by classifying the selector under the specified protocol in its ClassOrganization and then adding the new method to its method dictionary. As with source storing, this behavior from the old compilation message is better performed by the ClassDescription itself than by a MethodProducer operating on it.

PARAMETERIZING THE COMPILER

ParameterizedCompiler is quite similar to its superclass, SmalltalkCompiler. It adds accessor methods to set the value

of each of the classes used in the compilation process. It uses five Factory Methods to instantiate its co-workers, providing a locus for overriding in future subclasses. Instances of Parser, ProgramNodeBuilder, and MethodNodeHolder are instantiated simply by sending #new to the class. The other two methods are:

```
newCodeStream
    ^codeStreamClass new owner: self

scopeForClass
    ^self nameScopeClass forClass: class
```

These implementations show why Factory Methods sometimes produce classes and sometimes produce instances: because different "factories" (i.e., classes) sometimes need different messages and associated information to produce valid instances. ParameterizedCompiler overrides the key worker method of step 4, #translate:noPattern:ifFail:needSourceMap:handler:, as follows:

```
translate: aStream noPattern: noPattern ifFail:failBlock
needSourceMap: mapFlag handler: handler
    | methodNode holder codeStream method |
    methodNode :=
        self newParser
            parse: aStream
            class: class
            noPattern: noPattern
            context: context
            notifying: handler
            builder: self newBuilder
            saveComments: mapFlag
            ifFail: [^failBlock value].
    methodNode := producer transformTree: methodNode
in: class.
    handler selector: methodNode selector. "save selector
in case oferror"
    codeStream := self newCodeStream.
    codeStream class restartSignal
        handle: [:ex |
            codeStream := self newCodeStream.
            ex restart]
        do: [codeStream class: targetClass outerScope:
selfscopeForClass;
                requestor: handler.
            mapFlag ifTrue: [codeStream saveSourceMap].
            noPattern
                ifTrue: [methodNode emitValue:
codeStreaminContext: context]
                ifFalse: [methodNode emitEffect:
codeStream].
            method :=3D codeStream makeMethod:
methodNode].
    holder :=3D self newMethodHolder.
    holder node: methodNode.
    holder method: method.
    mapFlag if True: [holder sourceInfo: codeStream
sourceInfo].
    ^holder
```

There are two things to note here in contrast to the superclass' implementation of this method. First, no classes are referred to by name—they're all accessed by the instantiation methods shown earlier, so that they can be easily changed and specialized. Second, there is a new message send just after parsing, which is our second transformation stage. (In a similar way, messages can be added at the other stages so that the compiler's producer can transform the objects flowing through the production points.) The producer working with this ParameterizedCompilier is given the newly obtained parse tree and the class it is being compiled in with the #transformTree:in: message. The producer is expected to return the parse tree that should be used for code generation. By default, MethodProducer will simply return the parse tree that is the first parameter of this message, but in some cases (including when breakpoints are present), it will need to be able to manipulate the parse tree.

A parse tree, as returned from the parser in step 4a, is represented by its root node, an instance of class MethodNode. That node, and all others in the tree, are subinstances of the abstract superclass ProgramNode, which defines the common behavior of all members of a parse tree. Each superclass of ProgramNode has a different instance layout. For example, a MethodNode contains a single block as its child in the tree, while a messageNode

has both a receiver and an array of arguments as its children. This makes it hard to enumerate over a parse tree.

Fortunately, help is provided in the form of ProgramNodeEnumerator, an abstract class that outlines the procedure for enumerating over a parse tree. ProgramNodeEnumerator is an instance of a common design pattern, in this case one called Visitor. The Visitor pattern represents an object that operates on each object in a complex, heterogeneous structure, performing some function on each member that is dependent on the type of the member. However, ProgramNodeEnumerator is only an abstract representation of a Visitor: it doesn't actually do anything when it "visits" the various nodes in a parse tree. We implement a concrete subclass in the form of ProgramNodeEvaluator, a class that provides the normal block-based enumerations used in the Collection classes, including #do: and #select:.

ParameterizedCompiler allows programmers to override behavior at each production point and transformation stage, providing great flexibility in the compilation process.

The ProgramNodeEvaluator interacts with the ProgramNodes using a technique called double-dispatching: the evaluator sends the generic #nodeDo: message to each node with the evaluator itself as a parameter, which in turn sends back a class-specific message to the evaluator, such as #doMessage:receiver:selector:arguments: from a MessageNode and #doAssignment:variable:value: from an AssignmentNode. The evaluator implements these messages by sending itself #doNode for each node passed to it; in the case of the AssignmentNode, that would mean the variable node and the value node, while in the case of the MessageNode it includes the receiver node and all the argument nodes. Finally, the evaluator implements #doNode: by evaluating the block with the programNode parameters as an argument and then sending #nodeDo: back to that node to continue the enumeration. The enumeration terminates because the various leaf nodes, such as LiteralNode or VariableNode, have no ProgramNode children for the evaluator to #doNode: on.

There is one final issue to consider before our implementation is complete. Prior to our work, parts of the system could compile a method by sending appropriate messages to its target class, or they could parse or compile methods by sending messages to an instance of a Parser or Compiler obtained by the class' #compilerClass message. Now, however, it is important that all method parsing and compiling be done through the appropriate MethodProducer, so that any extensions or variations in the compiled code are handled consistently and correctly. As a result, we have to search through the image to find all places where Parser and Compiler are used directly to see if they are still correct. If they are not, we must change the site to use either the class or its MethodProducer.

We use the term "refactoring" to describe a common process that developers use to reorganize a program without changing its basic functionality—examples include introducing abstract classes, renaming variables and messages, changing inheritance relations to composition relationships, and so on. Programmers refactor programs to make them easier to reuse, maintain, and understand. Refactorings are a new and useful way for programmers to think and communicate about what they do, and they also establish a basis for developing tools to support refactoring work. In our experience, the process described in the previous paragraph is a common refactoring applied while doing system rework, and we call it Trail Splitting. When our system rework introduces one or more new ways of thinking about an existing process, it splits one existing trail into many. We must then find each place where the trail forks and point the way (by changing code as necessary) down the correct path. We also applied this refactoring in the implementation of breakpoint methods (as we'll see next issue) and when we introduced lightweight classes.[1] For the latter, we created a new message, #dispactchingClass, which returned the first class in an object's look-up chain, as opposed to the existing #class message, which returned the class used to instantiate the object, define its layout, and many, many other things. Any method that had previously sent the message #class to obtain information now returned by #dispatchingClass had to be changed.

CONCLUSION

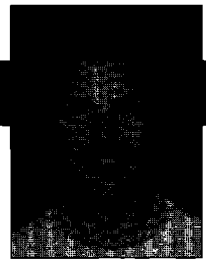
This concludes the implementation of a new, more flexible compilation framework. Next issue, we will use the MethodProducer–ParameterizedCompiler pair to implement breakpoints. In the process, we'll see how this new subsystem increases the environment's programming flexibility as well as some extensions that allow different MethodProducers to be combined.

Authors' note

Source code for the parameterized compiler is available by anonymous ftp from st.cs.uiuc.edu. Look for the file ParameterizedCompiler20.st in pub/st80_vw (or ParameterizedCompiler41.st in pub/st80_r41 for ObjectWorks4.1 support).

References

1. Hinkle, B., V. Jones, and R.E. Johnson. Debugging objects, The Smalltalk Report 2(9), 1993.
2. Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.



Kent Beck

A modest meta proposal

I JUST GOT AN ISSUE OF SMALLTALK REPORT that had someone's written summary of one of the talks I gave at Smalltalk Solutions. I sound like a wild-eyed, fire-breathing, spiky-haired maniac! It is so strange to see how others see me, especially in public. I'll admit to being in rare form in New York, a little over the top on the outrageous meter, but really...

The other shock this month was news of the ParcPlace/Digitalk merger. I see the press release. I check the date. Nope, not April 1. Hmmm...Is this some kind of elaborate joke (badly timed and in extremely bad taste)?

Now that I'm over the shock, I can see positives and negatives in the deal. It makes sense for Digitalk because (as Robert Yerex from ObjectShare pointed out) they got a much better valuation than they would have on the open market. It makes sense for ParcPlace because their worst nightmare was Digitalk's technology married to somebody with cash and marketing clout.

The outlook for customers isn't so one-sided. If all goes well, the current products will get their holes filled. VisualWorks will get native widgets and better performance. V will get a better garbage collector and fuller application model. Digitalk's culture of getting stuff out the door married with ParcPlace's culture of striving for elegance could be a potent brew. On the other hand, if sales aren't going well there will be a lot of pressure to drop one or the other image before PPD can architect an orderly transition.

All this spells opportunity for the other vendors to invoke that good old FUD factor, and pick up some quick market share. They'd better, because if they don't and PPD starts hitting on all cylinders—look out!

CLIENT: OOCL

I've gotten several questions about what it's like to be a consultant. By the time this is published, everyone on the planet who knows how to write Smalltalk may already be a consultant, but just in case, I thought I'd provide a short sketch of one of my clients and what I do for them.

Orient Overseas Container Ltd. (OOCL hereafter) is a $1.5 billion (US) global container shipping company headquartered in Hong Kong. Their business is delivering those standard-sized containers you see pulled by trucks on the highway from point A to point B, where A and B could be anywhere in the world. They own or lease hundreds of thousands of containers and chassis. They operate 30 some container ships. They run terminals, depots, and transshipment yards all over the world. They interact with hundreds of thousands of customers, all of whom rely on OOCL to get shipments delivered on time. They handle more than one million shipments per year.

While these aren't numbers to impress Federal Express (with a peak of three million shipments per *day*), they are pretty respectable, especially when you factor in the tremendous amount of capital involved in the form of containers, ships, and yards. Container shipping is heavily regulated worldwide, so small reductions in cost or improvements in productivity make a huge difference on the bottom line.

OOCL's current IS operation is centralized in Hong Kong, built around a large IBM mainframe. To gain flexibility, reduce cost, and better address local requirements (imagine having to satisfy a hundred different customs bureaucracies with one system), they decided to move to a more distributed, client/server system. They chose Smalltalk (VisualWorks) for the front-end implementation language.

The project, IRIS-2, is medium-scale by IS standards. They plan to have around 40 developers when things are in full swing. They located in Silicon Valley to be closer to Smalltalk talent.

I've been involved with IRIS-2 since it began its life in these United States. I've had a number of jobs as the project has matured:

- At first we were all just trying to figure out the architecture, so I was a design consultant. We slung CRC cards, acted the part of objects, and learned about each others' specialities.
- As the design became clearer, David Ornstein and I wrote an architectural prototype of a critical part of the design so we could be sure we weren't just making beautiful diagrams.
- I helped design and deliver the "Smalltalk Boot Camp," a three-day simulation of the entire software lifecycle intended to bring teams closer together and promote good programming practices.
- OOCL has generously sponsored my pattern writing,

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

using the Smalltalk Best Practice Patterns I have been working on as part of their developer guidelines.
- I have been visiting about twice a month all along to review code, suggest improvements, and tune performance.

I have learned a number of interesting lessons for myself and for projects like this, which are becoming the norm in the Smalltalk world. On my part, I have learned:

- **A stand-up lecture is useless for teaching.** I have given a series of lectures about patterns that seemed to have no impact. To address this, we held a "Pattern Bowl," where teams were challenged to find patterns or the absence of patterns in existing code. I think everyone learned more in those two hours than in tens of hours of lecture before.
- **Be outrageous.** What we are doing is difficult. It is risky (does anyone know the source of the factoid that 50% of all software projects never deliver?) There is a lot at stake. Plodding along in a humdrum way doesn't cut it. If I want to have impact I have to go for risk and flash, not "just the facts." The Pattern Bowl is a good example. We had prizes, applause, an obnoxious timekeeper (me), tension, competition, and the all-important fuzzy animal to go into the keeping of the winning team. Hokey? Yes, but it works.
- **Don't be too hard on yourself.** A consultant can only do so much. In the end, the success of the project isn't my responsibility. I'm responsible for doing the best I can, and suggesting other things that I can see need doing. When a deliverable slips, it doesn't help to get caught up in the emotion. It's hard to care but not too much, but that's what it takes to be effective.

This project has shown me that Smalltalk has some serious holes. I have been swimming in Smalltalk for so long that I no longer see the water. Newcomers to Smalltalk find it anywhere from irritating to impossible. For the market to grow, the vendors absolutely have to address the issues raised by new Smalltalkers.

For projects, I learned:

- **Baby steps.** Do one small thing, then one slightly larger thing, and on and on. The temptation to jump in with both feet is overwhelming. The argument always goes "I have committed to this date. I can't do it with baby steps. I have to ramp up more quickly." The result is always disaster. Always. OOCL has done a good job of trying to stick to baby steps and of getting back to baby steps when they have gotten too big too fast.
- **Program in pairs.** The most productive form of programming I know (functionality/person/hour) is to have two people working with one keyboard, mouse, and monitor. Our educational system trains us not to do this and some upper managers have a hard time with it ("Why did we buy all those workstations and cubicles if we don't use half of them?"), but it makes a bigger change in productivity than any other single change.
- **Follow standards.** There are two parts to this. First, you have to have standards. In writing patterns, I'm deeply embroiled in exactly what the standards should be, but honestly, it is far better to have adherence to good standards than deviation from perfect standards. Second, you have to follow them. OOCL has recently put in place a schedule of peer review that makes sure everyone's code is seen by a critical audience at least every couple of months. This ensures that everyone has a motivation to understand and follow the standards, if only to avoid being ripped in public.

There's a lot more, both to the project and what I've learned, but it will have to await another column. I'm running out of space and I still haven't gotten to my technical topic...

A MODEST META PROPOSAL

"Meta programming? Isn't that what PhD's do to get thesis? What does that have to do with getting my next deliverable out?" Even if you don't know it, you're probably already doing some meta programming. Meta programming is writing programs that manipulate not your objects, the way usual programs do, but the *representation* of your objects. For example, the fact that each object has a hidden "class" instance variable, and you can fetch any object's class and ask it interesting questions, is meta programming. `IsKindOf:`, `respondsTo:`, `instVarAt:`—these are all messages about how the receiver is represented.

Smalltalk makes meta programming easy. Too easy, in fact. When you meta program, you are no longer really programming in Smalltalk, you are inventing a new programming language that is an extension of Smalltalk. Used indiscriminately by application developers, meta programming is a disaster. Just as not everyone can write reusable software, not everyone can write new programming languages. When everyone is writing in their own Smalltalk increment, and all the increments are different, disaster lurks. You can no longer read a line of code and guess what it does correctly. Risk soars and so does the cost of maintenance.

On the other hand, the meta programming facilities of Smalltalk can come in extremely handy. They can even save a project. If having some new kind of control structure vastly simplifies your program, chances are you can implement it in Smalltalk and take advantage of it.

How, then to provide the needed facilities without exposing them unnecessarily? The problem as I see it is that they are all implemented up there in `Object`. It's just too easy to stumble across `isKindOf:`, use it to solve a short-term problem, and never discover the powerful polymorphism lurking just around the corner. I propose to put up a wall between application programmers and meta programming by introducing a new class, `MetaObject`, upon which all the current meta protocol in `Object` (and some in `Behavior` as well) will be heaped.

This is not an original idea. I got the idea in 1987 from Patti Maes' OOPSLA paper. I don't remember the exact title any more, but it introduced the idea of meta objects.

I've had the idea floating around in my head since then, but I didn't do anything about it until I was bored on a flight recently. Pulling out my trusty ThinkPad, I whipped together an implementation. I liked the result enough to publish it here.

MetaObject is an Adaptor on any object. An Adaptor changes the protocol that an object accepts by interposing an object with the changed protocol.

```
Class: MetaObject
    superclass: Object
    instance variables: object
```

You create a MetaObject by giving it the object to adapt:

```
MetaObject class>>on: anObject
    ^self new setObject: anObject
MetaObject>>setObject: anObject
    object := anObject
```

There is a Facade in Object, Object>>meta, for creating a MetaObject. Clients will use this interface.

```
Object>>meta
    ^MetaObject on: self
```

The infamous isKindOf: becomes "inheritsFrom:" in MetaObject:

```
MetaObject>>inheritsFrom: aClass
    ^self objectClass includesBehavior: aClass
```

ObjectClass replaces Object>>class:

```
MetaObject>>objectClass
    ^self object class
```

I don't have space here to show all the implementations of the MetaObject protocol. Table 1 shows the old and new meta protocol. In some cases, I'm not thrilled with the new names. I'll happily entertain suggestions for better selectors.

This is certainly not an exhaustive list. It's just what I came up with in a couple of hours. It should be possible to move more meta programming protocol in MetaObject.

Given this amount of protocol, I was able to quickly produce an Inspector that used a MetaObject to display and modify instance variables.

MetaObject provides the following advantages:

- **It discourages casual use of meta programming protocol.** If you see "meta" in application code, you'll know to perk your ears up and make sure it really belongs.
- **It collects scattered protocol.** Some j4

meta programming protocol is implemented in Object, some in Behavior, some in Class. MetaObject brings it all together in one place.

- **It is flexible.** If a particular class needs a different kind of MetaObject for some reason, it can override "meta." You might do this, for example, to give a uniform programming environment on Smalltalk and C++ objects.
- **It simplifies Object.** Let's face it. Object is too darned big. VisualWorks 2 (the Envy version, anyway) defines 166 methods on Object. Visual Smalltalk Enterprise 3.0 defines 348. IBM Smalltalk gets by with 101. MetaObject is a step in the right direction.

MetaObject has the following disadvantages:

- **One more class.** Don't we have enough classes in the base system already? We will have to teach people to use it and convert old code.
- **One more object.** Now, when you want to have access to meta protocol you have to create a whole new instance of MetaObject.

How about it? Next time you need meta programming, implement a little MetaObject first and see how it feels. Let me know if you like it.

Table 1. Old and new meta protocol.

Object meta message	MetaObject message	Explanation
class	objectClass	Return the class the receiver instantiates.
changeClassToThatOf: aClass (VisualWorks)	objectClass: aClass	Change the receiver's class.
class allInstVarNames	keys	Return the named instance variables (MetaObject lets you treat an object like a Dictionary).
class allInstVarNames size	size	Return the number of named instance variables.
instVarAt: aNumber	at: aString	Return the value of an instance variable.
instVarAt: aNumber put: anObject	at: aString put: anObject	Change the value of an instance variable.
allOwnersWeakly: aBoolean (VisualWorks)	owners	Return a Collection of all objects refering to the receiver.
become: anObject	switchWith: anObject	Swap two objects identities.
isKindOf: aClass	inheritsFrom: aClass	Return whether the receiver inherits from aClass.
isMemberOf: aClass	instantiates: aClass	Return whether the receiver is an instance of aClass.

Rules to live by

Mark Lorenz

I WAS RECENTLY ON A PANEL DISCUSSION at ObjectWorld Boston about problems O-O software projects encounter and how to recover from them or, better yet, avoid them in the first place. It got me to thinking about the lessons we've learned and how they keep coming back over and over again. It's been awhile since I listed a set of these topics, so here goes.

ERRORS AND RECOVERY

There are a number of problems we could discuss, certainly too many to exhaustively list here. So in this section I'll list some of the problems I see most frequently on the projects I work on.

Missing model

This problem results when you connect a graphical user interface (GUI) directly to your existing database (DB). This design keeps you from achieving reuse and lower maintenance costs when developing your software systems. Object technology's great potential is primarily achieved by developing and leveraging a model of your business domain. This model is surfaced through the GUI. A database is merely persistent storage underneath the object model.

If you need a quick ad hoc solution to some need in your organization, you can by all means slam together an application that is all GUI and DB. Just don't kid yourself into believing that you will have an easier time reusing, maintaining, and extending the application over time.

Some of the development products available today make leaving out the model between a GUI and DB very easy to accomplish. Interfacing objects to a relational database (RDBMS) used to require a "broker" layer of software to handle the data movement to and from object state data and database rows. IBM's VisualAge is an example of one such product. VisualAge handles many of the details of accessing RDB row data. In fact, as Figure 1 shows, you can make direct connections from GUI widgets to database row information.

Mark Lorenz is founder and president of Hatteras Software Inc., a company that offers services and products to help other companies use object technology effectively. He welcomes questions and comments via e-mail at mark@hatteras.com or phonemail at 919.319.3816.

Staffing behemoths

This problem occurs when you have offices full of people and the project is just starting to develop an object model. It occurred on a project of mine a few years back. Another modeler and I showed up to begin developing an object model. We were shown around the group and discovered that there were over 25 developers and all the surrounding support staff on the project. We gathered a couple of technical leads and a couple of domain experts and the six of us went into a conference room to start the rapid modeling sessions. I asked the woman who eventually became the de facto chief architect on the project "what are all the other people doing while we're in here?" The answer was "they have things to do." Well, they basically wasted time waiting for us to get far enough along with a model and subsequent architecture of subsystems and contractual interfaces. They were then put to good use.

My previous work discusses how to architect your system so that teams can work relatively independently and still be productive in building a cohesive system.[1,2] Using these techniques, you can effectively grow your organization and avoid the costly mistake of staffing too many too soon.

Ill-behaved object model

An "object model" that has all data and no behaviors is a typical indication of this problem. A group from a telephony project once proudly marched me into a room to see their object model pasted onto a wall. The pages and pages

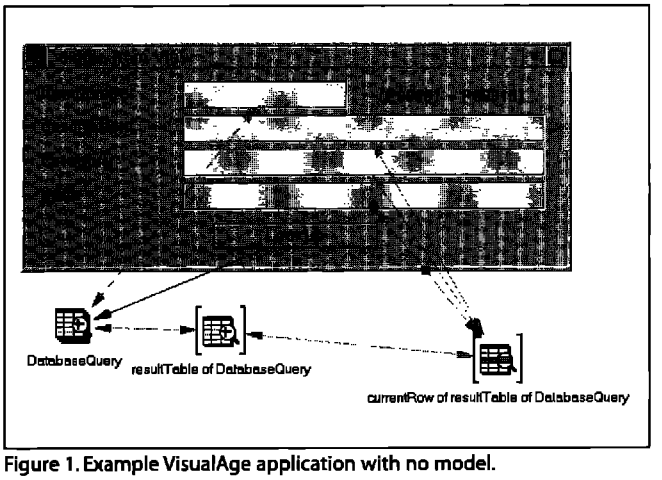


Figure 1. Example VisualAge application with no model.

of output included every imaginable piece of state data that could be associated (and had been in their legacy database!) with the classes they had identified. And not one class had a single behavior in it! Figure 2 shows an example of what this type of model looks like. You can spot it from across the room, even if you can't read the details, because the middle state portions of the class boxes are filled with text and the bottom behavior portions are empty.

An object model must focus on behavior, as shown in Figure 3. I've purposely left all state data off this diagram to drive the point home that behaviors and their allocation are essential to success. Certainly, you will want to (eventually) show state data in your object model.

Missing management

There are differences in managing a team developing O-O systems using different processes and methodologies rather than traditional techniques. Managers need training in what to track, what it means, how to schedule, how to organize teams, and so on.

Figure 4 shows an example of basing schedules on use cases, scenario scripts, and subsystems. This is different than traditional schedules, which are generally based on functional line items. This new type of schedule has dependencies on the team organization also. For example, most of the time one small team will work on one subsystem start-to-finish. This requires that work on business scenarios that affect their subsystem be scheduled serially along with other subsystem teams. Support subsystems can be worked on independently, as long as they are ready before dependent business scenarios need them.

Persistence black hole

I have seen whole projects eaten alive by this problem. There are various facets to integrating O-O systems to legacy systems and databases. It requires brokering to map between the object's state data and the RDB rows, as shown in Figure 5. This is often just the tip of the iceberg, however. When you start getting into issues of distributed objects, shadow objects, system startup and shutdown, and error recovery, the situation gets much more complicated.

Basically, you end up spending a large percentage of your time getting into the object database (ODBMS) and support tool businesses. You worry about how to handle long DB transactions, rollback, and other issues. This obviously takes time away from your real business, such as building a finance, insurance, or retail application.

Depending on your requirements and the products available when you encounter this beast of a problem, you have different options. An easy one, if it meets your needs, is to use an ODBMS such as GemStone, Versant, or ObjectStore.

AN OUNCE OF PREVENTION

Preventative measuring

O-O metrics can assist you in various ways, from developing better estimates for new projects to checking on the quality of projects already underway. The goal is to find and resolve problems as soon as possible.

The OO metrics that give you the most "bang for the buck" are organized as follows:

- Method size—number of message sends
- Class size—number of methods and variables
- Coupling—law of Demeter, global usage
- Inheritance—method overrides, hierarchy nesting depth
- Complexity—McCabe for classes

See OBJECT-ORIENTED SOFTWARE METRICS[3] for a complete discussion of each of these.

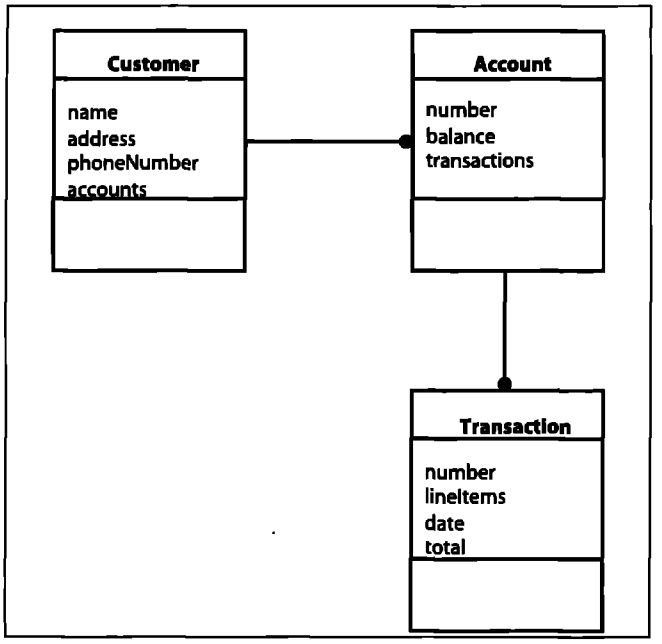


Figure 2. Example of a data model mistaken for an object model.

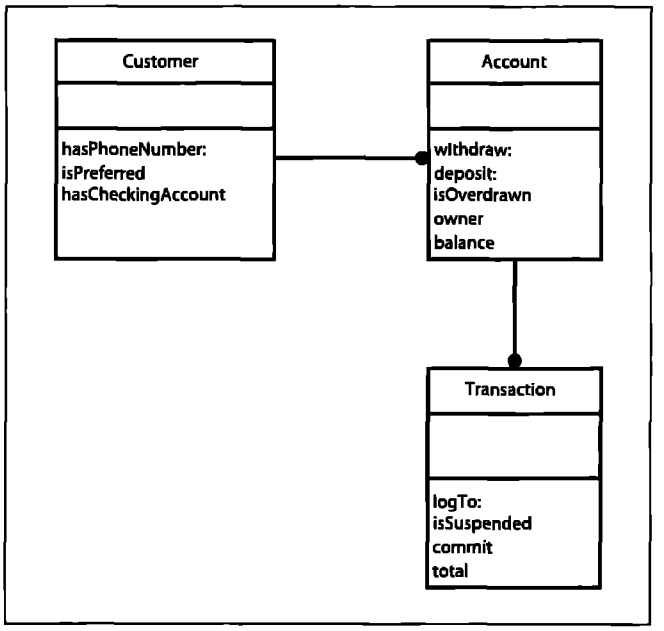


Figure 3. The same business represented by a true object model.

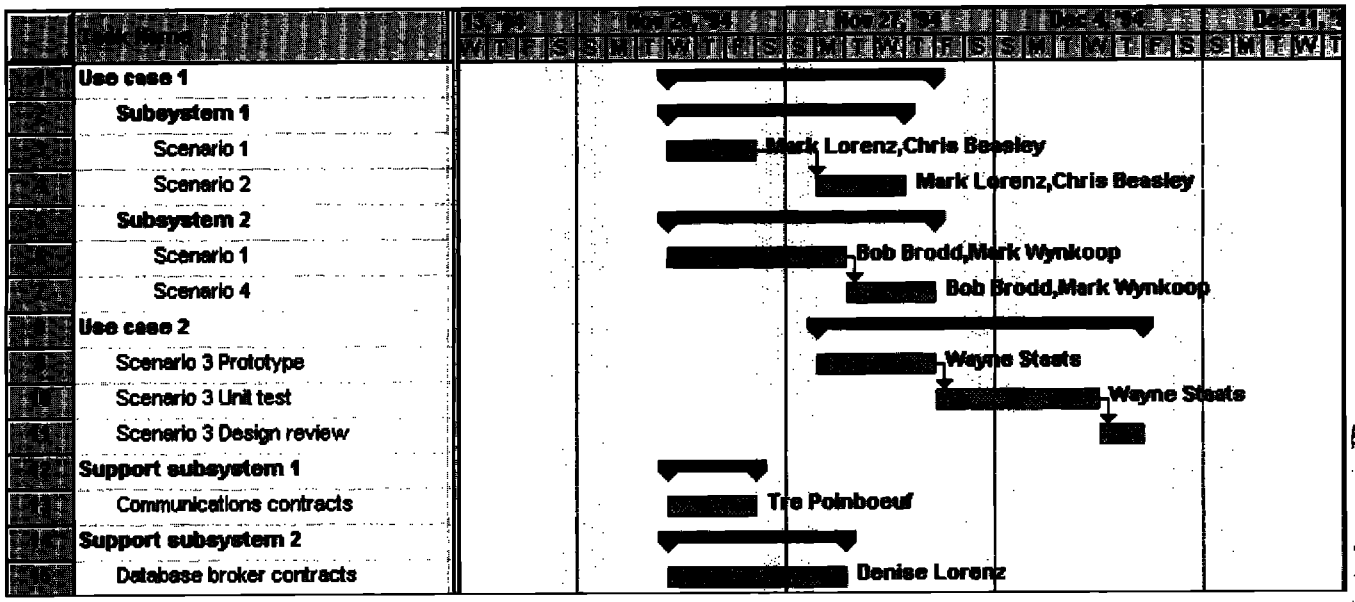


Figure 4. Example schedule based on use cases.

Design, don't just code

Most project teams focus on coding issues, such as language syntax and tricks, instead of what's really important—the object model and design conventions. I recommend the following techniques for your design (discussed in greater detail in RAPID SOFTWARE DEVELOPMENT[2]).

Instantiation integrity. This technique ensures that your model state is valid at all times through the use of custom class instantiation methods. For example, if your business rules require a SalesTransaction to have a Customer associated with it, you might have a class method such as:

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

    ^self new
        customer: aCustomer;
        yourself
```

Collection protection. This technique protects your state from mistakes made by your clients by passing them copies of your information. For example, if a view class asks the Store for its employees, a copy of the Collection is returned so that the real Collection cannot be corrupted.

```
Store
employees
        "return a copy of my employee collection"

    ^self myEmployees copy
```

Laissez-faire initialization. This technique makes your objects more robust by having them self-initialize as needed at runtime. It also allows for business rule enforcement and ease of redesign because you have a point of control for state access.

```
Store
myEmployees
        "Private: return my collection of employees"


    ( myEmployees isNil ) ifTrue: [ self
        myEmployees: OrderedCollection new: 10. ].
    ^myEmployees
```

Invest in an object model

The most important O-O software asset is your business' object model. It is absolutely essential to your success that you spend time developing a model of your business concepts, relationships, and service requests before design and implementation. Get O-O and domain experts in a room, write use cases and scenario scripts, and draw object model and collaboration diagrams.

Get mentoring

The fastest way to get your people over the learning curve is through mentoring. There is no replacement for direct interaction with people who have developed O-O systems before. Developing a good O-O system takes a lot more than a language class!

Run your project like a group of small projects

The Standish Group did a study of 8,380 applications and found that 78% of small company software projects were successful, whereas only 9% of the large company projects were successful.[4] The message to me is that the only realistic way to run a large project is by dividing the team up into relatively independent smaller teams.

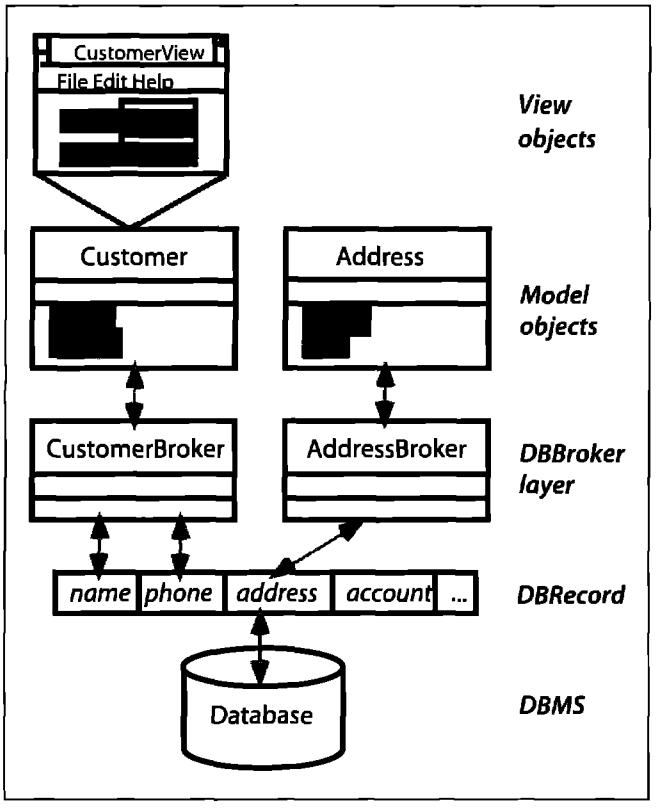


Figure 5. Use of a DBBroker.

SUMMARY

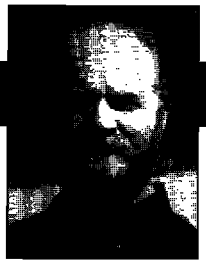
We've gone over a number of the most common problems we run into on O-O projects. We've discussed ways to resolve them when they happen and, more importantly, how to avoid them in the first place.

Terminology

- **behavior:** The services provided by an object to other objects, through messaging and method invocation.
- **collaboration diagram:** Graphical representation of the subsystem groupings of classes and the contractual relationships between subsystems and key classes.
- **object model:** Objects and their relationships required to represent your business domain and business rules.
- **distributed object:** An object that resides on another processor.
- **object database:** A persistent store that works seamlessly with object definitions and/or instances.
- **shadow object:** An object that has a proxy stand-in on the local processor, but actually resides on another processor.

References

1. Lorenz, M. Architecting large projects, THE SMALLTALK REPORT 4(6):28–29, 1995.
2. Lorenz, M. RAPID SOFTWARE DEVELOPMENT, SIGS Books, New York, 1995.
3. Lorenz, M. and J. Kidd. OBJECT-ORIENTED SOFTWARE METRICS, Prentice Hall, Englewood Cliffs, NJ, 1994.
4. The Standish Group International. CHAOS, 1994.




Jan Steinman Barbara Yates

Managing project documents

IN OUR PREVIOUS COLUMN, we made a case for "continuous documentation," and outlined what that entails. We also promised to give you some concrete examples and source code, so you could begin to implement a continuous documentation process.

First of all, we'll need to change how classes store their comments...WE INTERRUPT THIS COLUMN TO BRING YOU A BASE IMAGE CHANGE ALERT! ALL USERS WITHIN 200 KILOBYTES OF THE IMAGE MUST EVACUATE IMMEDIATELY! WHEN YOU ARE ALLOWED TO RETURN, YOUR PRECIOUS, CAREFULLY CRAFTED, WORK-OF-ART CODE WILL TAKE ON STRANGE AND (we hope) WONDERFUL NEW BEHAVIOR! HAVE A NICE DAY!

Whew! We almost slipped one by the Base Image Police there, but they caught us! So, let's retitle this column and proceed.

MANAGING MODIFICATIONS (OR "WHO CHANGED basicNew?")

Pity the poor Smalltalk vendors! You buy an object library in C++, and you typically get linkable object code—it works, or it doesn't. But when Smalltalk customers don't like what they got from their vendor, they simply change it—which often introduces bugs, which are often subsequently reported back to the vendor! (All of this applies to third-party code as well.)

Consider the myriad ways that basic Smalltalk can become polluted:

- **Beginner naiveté**. "Delay := Delay forSeconds: 1."
- **Enough knowledge to hurt yourself**. A seasoned ST/V user tries VisualWorks, and writes a cleanup method that does "MyClass allInstances do: [:inst | inst become: nil]."
- **Enough knowledge to make it look random**. The same code as above, but cleverly made conditional upon rare low-memory conditions, and then forgotten.
- **Unintentional overrides**. Such as implementing nextPutAll: in a Stream subclass that normally inherits it.
- **Forgotten halts and other test or debug code**. Beginners often put halts in system code (rather than putting halts in their own code, then stepping *into* the system code), and sometimes they forget to take them out.
- **Well-meaning changes gone awry**. Such as the datacom specialist who changed *Integer* printOn: so that if the shift key is held down, they print in hexadecimal. (This one didn't quite make it to production before someone noticed strangeness when extending selection in a table by shift-clicking...)
- **Downright malicious**. Nah, no Smalltalker would make malicious changes, right? But if someone did, say a disgruntled soon-to-be former employee...

Always keep in mind that base changes are the enemy of reuse. One of the big wins of reuse is that less testing is needed when you reuse previously tested code. The down side is that changing code that is heavily reused *increases* the testing burden, because you aren't really sure all the uses of the changed code agree with each other.

Why are changes necessary?

In team programming, base changes fall into two categories. *Personal changes* are necessary for individual developers. Individuals need to be able to experiment with base changes before foisting them on their teammates; they may experiment with base changes to better understand the environment; or they may simply want to customize their own environment. If you are using a code management system (such as ENVY or Team/V), you generally have numerous options for balancing the needs of the team for stability against the needs of the individuals for experimentation.

The second category is where the trouble begins. Although you should do whatever you can to discourage it, sometimes you need to make *project- or corporation-wide* base image changes. These changes might include:

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

- **Fixing bugs in vendor's code.** This is fairly unusual, but if you do find a bug that is getting in your way, and it has an obvious fix, you will probably want to incorporate it into your base. Also, maintainers *love* getting bug reports with fixes, so if you fix the bug carefully, document it properly, and submit it with your bug report to the vendor, there's a good chance it will be in the next release from the vendor, which makes your re-integration job that much easier.
- **Make enhancements to the vendor-supplied tools.** This is the category for which the Base Image Police caught us! The combination of dynamic compilation and full source code means you can easily tailor the Smalltalk development environment to your organization's specific needs. These kinds of changes have little possibility of getting into a vendor's product, and so they must be done in such a way that facilitates re-integration with future vendor releases.
- **Make enhancements to the vendor-supplied framework classes.** This is similar to the previous case with one important difference: the changes you make to framework classes will be delivered with your application, and so must be more robust than changes made to development tools. This should involve regression testing to ensure that the framework still functions with previously written code.

"When Smalltalk customers don't like what they got from their vendor, they simply change it."

Limit scope and impact of changes

Base image changes can be categorized by their scope. You should carefully analyze your needs, and limit the scope of your change to the greatest degree possible. For example, it may first seem that you need to add an instance variable to a base image class and add two methods that use that new state, but further analysis might show that you really only need to change the use of an existing instance variable, and then hide that change by changing the methods that access that instance variable. The following change categories are roughly in order of desirability.

Single-method, non-state changes are the best. Such changes should not change the arguments or answer of the method, but only its side-effects. The answered object should have the same behavior, and no additional constraints should be placed or assumed on variables or sent methods.

Encapsulated state changes put different objects in instance variables, but manage those changes through the methods that access those variables. These changes tend to have small impact on any given vendor release. For example, you might need something better than simple truncation of window labels to use as icon labels, so you could change the label instance variable to be a two-element *Array* that either answers a full, title-bar length label if the window is open, or a custom short label if the window is iconified.

Another useful encapsulated state change is arranging for an instance variable that normally holds a method selector so that it can hold a block. This can be a useful change to "pluggable views" for increasing the dynamic behavior of your system, and if properly done, is essentially invisible to old code.

A big problem with this technique is that object state is directly visible to subclasses. If some poorly written subclass directly accesses the state you have changed, rather than going through the access methods you changed in tandem, there will be trouble, and it may be difficult to diagnose.

Method overrides don't seem like changes, but they can have tremendous impact. (If you don't believe us, override *Behavior* basicNew with a new implementation in the *Object* class, then purposely introduce a bug and see what trouble *that* causes!)

Overrides are tempting, because they do not change actual base image code, but for that same reason an override is difficult to track and debug. They are more trouble when re-integrating new vendor releases—none of your comparison tools will detect the override as a change, but it may well conflict with vendor changes in the new release.

Finally, there is usually a reason for the inheritance of such methods—if an override seems attractive, be sure that the change shouldn't actually go into the inherited method.

Changing message arguments or return objects begins to get messy, and should be avoided. Constraining arguments or returned objects, such as requiring that an argument be an *Array* rather than any kind of collection, might work for your particular case, but it is certain to eventually break someone else's code that didn't share your assumption.

Changing object shape, or the number or ordering of instance variables, is one of the most invasive changes you can make, and is to be avoided. Adding instance variables by itself is not terrible, but if such a change is really necessary, most of the time it is because the fundamental behavior of the class is being changed—behavior changes are what subclasses are for!

Changing interclass interfaces is really dangerous. You might track down all uses in your context, but third-party software won't know about your change, and your Smalltalk vendor's next release certainly won't know either! If changes this extreme are required, be certain to document them well, to ease the inevitable problems that eventually will result.

"Conditionalize" changes

Especially in the latter categories mentioned above, it becomes increasingly important to factor your change in

continued on page 31

Recruitment Center

The American Funds Group®

The American Funds Group is one of the most successful mutual fund organizations in the world. Since 1931, we have provided our shareholders with consistently superior investment results and outstanding service. Share in the continued growth of our Norfolk, VA Office.

We have been a financial industry leader in Smalltalk development for over 5 years. We are currently developing a large client server based customer service system. This application is being created using the latest object oriented methods and is in the beginning stages of development. Ideal candidates will have the opportunity to be a part of the design team whose responsibilities will include these initial phases of development.

We offer a competitive salary and excellent benefits package including:

- Medical, dental and vision care coverage
- Educational assistance
- An outstanding company-paid retirement plan

Positions are currently available for:

Senior Smalltalk Programmer

This positon requires 2 to 5 years of Smalltalk experience including OOA and OOD. Job responsibilities will include leading in the overall design and creation of class and object hierarchies.

Smalltalk Programmer

In this position, you will develop GUI based client server applications. At least one year of Smalltalk experience is required.

If you are interested in applying for any of the positions listed above, please send your resume and salary history to:

The American Funds Group
(Please specify position)
5300 Robin Hood Road
Norfolk, Virginia 23513

EQUAL OPPORTUNITY EMPLOYER

KNOWLEDGE SYSTEMS CORPORATION

Make No Compromises. Join a leader in Object Technology.

We are Knowledge Systems Corporation, the acknowledged leader in Object Oriented Technology services. Working on the cutting edge of technology, we are poised to move to greater heights of technical diversity, client serviceability, and employer opportunity. We are professional, team oriented, and driven to excellence, but most of all, we are an employee-oriented corporation that provides an excellent working environment that will challenge your abilities and sharpen your skills. *We are KSC. We are your future.*

Presently, we are seeking to augment our technical training and consulting staffs with professionals who have two plus years of demonstrated experience with OOA&D, IBM Smalltalk or VisualAge, ParcPlace VisualWorks, Digitalk Smalltalk/V, and Envy.

As a leader in supplying our Fortune 500 client base with Object Oriented solutions, Knowledge Systems Corporation is able to offer a very competitive salary, an excellent benefits package and many opportunities to grow with the leader. Please send/fax your cover letter, resume, and salary requirements to: Knowledge Systems Corporation, 4001 Weston Parkway, Cary, NC 27513; or call (919) 481-4000; Fax (919) 677-0063 or e-mail to jdemichiel@kscCary.com.. Equal Opportunity Employer.

KNOWLEDGE SYSTEMS CORPORATION

SMALLTALK POSITIONS

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

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

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

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

DIGITALK

To place an ad in this section, call Michael Peck at 212.242.7447

ObjectSpace

Object Technology Professionals

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

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

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

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

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

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

IF OPPORTUNITY CALLS . . .

. . LISTEN, even though you're not "looking" now. Exceptional career-advancing opportunities for a particular person occur infrequently. The best time to investigate a new opportunity is when you don't have to!

You can increase your chances of becoming aware of such opportunities by getting your resume into our full-text database which indexes every word in your resume. (We use a scanner and OCR software to enter it.) Later, we will advise you when one of our search assignments is an exact match with your experience and interests, a free service to you.

Founded in 1974, we are a San Francisco Bay Area based employer-retained recruiting and placement firm specializing in Object-Oriented software development professionals at the MTS to Director level throughout the U.S. and Canada.

We would like to establish a relationship with you for the *long-term*, as we have with hundreds of other Object-Oriented professionals.

Lee Johnson International
Established 1974

Internet: lji@dnai.com URL: http://www.dnai.com/~lji
Voice: 510-787-2110 FAX/BBS(8N1): 510-787-3191
P.O. Box 817, Crockett, California 94525

ITT HARTFORD CORPORATE OBJECT GROUP

If you want to see the future, take a look at our past: 185 years of smart decisions have made us one of the few, true long-term success stories. That success continues today with superb ratings and bold new products, making ITT Hartford the smart decision for those with an eye on their future. We are currently seeking technical professionals to join our Corporate Object Group located in Hartford, CT.

OBJECT-ORIENTED BUILDER

The selected candidate will be responsible for the construction of corporate-level "infrastructure" object classes to provide utility functions and be leveraged by segment developers. You will review and harvest classes deemed appropriate for inclusion into the class library. Other duties include working with the Corporate Object Group and assisting project teams in developing classes to meet specific needs. Experience with C++, Smalltalk and relational database products is beneficial. Experience developing classes in an insurance environment and ability to integrate Smalltalk classes into ParcPlace VisualWorks are required.

OBJECT DATABASE SPECIALIST

We seek an individual to be responsible for object database tactical planning for the corporation. Duties include installing, configuring and maintaining an object database for the corporate class library and for group development and testing. Other tasks include implementing an object database network, performing product evaluations and providing resources for operational support personnel on ODBMS issues. Knowledge of C++ or Smalltalk and experience supporting object database products is required. Experience with GemStone, ObjectStore, Versant and relational technology coexistence and migration are a plus.

CORPORATE CLASS LIBRARIAN

This individual will assume responsibility for the administration of class specifications within the corporate object repository. Tasks include helping to define a system for easy navigation through the corporate library and managing multiple classes within the library. You will be a resource for library users and advertise available classes through internal publications. Experience in a large scale class library is desired. A working knowledge of C++, SmalTalk, ParcPlace VisualWorks Library and OTI ENVY is required. An understanding of the insurance business is a plus.

Please send resumes with salary history to:

Vincent Wright
ITT Hartford
Hartford Plaza
Hartford, CT 06115
Fax: (203) 547-2680

We are an Equal Opportunity/ Affirmative Action Employer

ITT HARTFORD

a way that makes it easy to back out. For example, if you want to add some special processing to what happens when you compile a method, it is tempting to simply put your modifications inline, but a better way is to make all your modifications in a separate method, then conditionally send that message if it exists, by using testing methods such as respondsTo: or canUnderstand:. Note that these tests can have a performance impact, but so can a broken change that you can't isolate!

This "base-change boundary" is one of the few places we tolerate the use of isKindOf:. Using isKindOf: as part of your program logic is contrary to good O-O design, because it imposes the sending method's viewpoint on another object rather than obtaining the other object's willing collaboration.

However, at the base-change boundary, isKindOf: is useful for testing the existence of base changes, so that they can be easily backed out without changing the base once again. It still isn't good O-O design, but it's a bit more justified when used to verify module interface boundaries.

Another useful technique for managing changes is to make them conditional upon an arbitrary "signature" method. For example, you might implement hasBeenHacked in *Object*, and then bracket your changes inside "(self respondsTo: #hasBeenHacked) ifTrue: [...]." This way, if a particular module of enhancements is present, they are used by changed base methods, but if not, the base changes skip the conditional changes.

Positive identification

There are two principle reasons to keep track of exactly what you changed: it will make your integration with the next new release from the vendor less painful, and it will help you to back out a change if it proves to be a mistake. Identification needs to happen at the method, module, and system levels.

For method changes, we've implemented a "hot key" that inserts "Modified by [user] on [date]: ." where "user" and "date" are properly filled in, and the cursor is positioned before the period to encourage the user to further describe the change. We use this two ways. In a short method, we simply place it after the normal method comment. In a long method, we bracket our changes by placing this hot-key comment both before and after the change.

Common code management systems allow version names for code modules. Keep in mind that your base image changes are not the main development stream; they are a branch! So if you modify a code component that the vendor named R1.43, you should not call your version R1.44 because that will most likely collide with the vendor's next release!

We use two techniques for naming changed base image modules; both help indicate a branch has taken place. The simplest is to append a "dot level," so the above example becomes "R1.43.0". This can get messy if you have a number of changes from different sources, so we often prepend some identifying information, such as "Bytesmiths R1.43.0." Either way makes re-integration with new vendor releases easier.

At the system level, a separate document that records every change or addition, organized by module, class, and method, is highly useful. These *release notes* are necessary even if your source code management system provides a version comparison tool; it is very useful to have a linear document to review when things start breaking!

"Keep in mind that your base image changes are not the main development stream; they are a branch!"

In some cases, your code management comparison facilities can be harnessed to survey changes and build templates for these release notes (we plan to demonstrate this in a future column), but documenting *why* and *how* the change was made will remain a human activity.

Organizational issues

In organizations with multiple Smalltalk development teams, there is usually an individual or a committee that has the authority to decide whether a particular change to the base image will be allowed. This role of base image "Keeper" is particularly important when there is a shared corporate-wide version of all base image classes.

A trial period for changes is a good idea. The Keeper cannot always tell that a particular change is benign to all the development teams' applications. If any team reports a problem with a change to the base image, the Keeper can then modify or back out the change to correct the problem.

Even when there is only one Smalltalk team, the integrity of the base image is usually guarded by a Keeper, who is the sole developer allowed to release changes to the base classes, also based on a trial period. In our experience, a trial of about one development cycle (six to eight weeks) is a good idea.

CONCLUSION

Now that you know how to make base changes in a way that is limited in scope, conditional, identifiable, re-integratable, documented, and "back-outable," we return you to your regularly scheduled column. In the next issue, we'll give you some actual base image changes to practice with as we proceed with examples of "continuous documentation."

JUST PUBLISHED!

Rapid Software Development with Smalltalk

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

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

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

About the Author...

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

Readers will...

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

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

PART OF THE ADVANCES IN OBJECT TECHNOLOGY SERIES

Available at selected book stores. Distributed by Prentice Hall.

SIGS BOOKS ORDER FORM

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

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

❑ Check payable to SIGS Books

❑ Visa ❑ American Express ❑ MasterCard

Card#_____ Exp._____

Signature_____

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

Name_____

Company_____

Title_____

Address_____

City/State/Zip_____

Country/Postal Code_____

Phone_____ Fax_____

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

SIGS BOOKS