

# Reuse As Reality

© 1999 by Prolifics, a JYACC Company, 116 John Street, New York, New York 10038, USA

All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express prior written consent of the publisher.

**For more information about Prolifics products, contact Prolifics at 1-800-458-3313 or via email at [sales@prolifics.com](mailto:sales@prolifics.com).**

Prolifics, JYACC, and JAM are trademarks of JYACC, Inc.

TUXEDO AND BEA ARE REGISTERED TRADEMARKS OF BEA SYSTEMS, INC. OTHER BRANDS AND PRODUCT NAMES APPEARING IN THIS DOCUMENT MAY BE TRADEMARKS OR REGISTERED TRADEMARKS OF THEIR RESPECTIVE COMPANIES.

# Prolifics: Reuse as Reality

## Introduction

“Reuse” has been the watchword of Object-Oriented Design and Development (OO) since its inception. This sounds like a valuable benefit; if we improve reuse, we write less code. Less code means faster development and easier maintenance in the future. Less code also means fewer chances for bugs, so it indirectly affects product quality. However, industry watchers report that there is only 15% average reuse in today’s object-based projects. That’s a pretty damning statistic. If true, we did better 20 years ago with COBOL subroutines!

Reuse, however, is not unique to OO. Good programmers have always attempted to reuse code through the use of structured programming techniques and “bottom up” design. This “informal” reuse was usually confined to single programs, programmers, or projects. Code was usually only reused across projects when programmers changed projects and took their code with them.

There is yet another form of reuse that we frequently take advantage of but we don’t normally think of as reuse. This is using the already-written code that is behind our development tools. In this paper I will discuss how Prolifics® provides extraordinary levels of reuse through a concept called Horizontal Components.

## The Challenge

Software development has become more difficult as requirements have become more demanding and systems have become more complex to meet these requirements. On the whole, complexity has increased faster than the methodology and tools have improved. With the advent of web-enabled transactional systems complexity has increased exponentially. Web-deployed applications must be massively scaleable, highly secure, and deliverable in “Internet years” of three to six months. To meet these requirements we are forced to move from 2-tier client/server architectures to 3-tier and multi-tier architectures. Instead of just a client and a server, we have a browser, a web server, an application server, database servers, and messaging middleware to tie it all together.

Most tools available today for multi-tier development are either 2-tier client/server tools, web-specific tools or application partitioning tools. Only Prolifics currently provides a tool that develops multi-tier, web deployed applications and handles the entire process from the web browser to the database. We will show how Prolifics does this by addressing all aspects of multi-tier computing; indeed, even the Prolifics tools themselves are inherently multi-tier, allowing multiple developers at remote sites to share resources on a server.

As a first step we should try to think of applications as composed of “components.” This concept is already gaining acceptance within the data processing community, but, as with any new technology, not everyone means the same thing when they refer to components.

## Components

Components are a step beyond “objects” (as meant within OO). They are not as well defined as objects are, however. There is a well-established theoretical basis for object-oriented methodology. Even if some developers don’t understand it, don’t use it correctly,

---

or disagree with it, there is a body of reference material that precisely defines objects and regulates their use. Components have no such pedigree, however.

When we refer to a “component” we do not mean another name for a widget (one common misuse of the word). We are looking for *business* functionality in components. A component is made up of business rules, application functionality, data, or resources that are encapsulated to allow reuse in multiple applications. In addition, components should be portable and inter-operable across applications, something that we will come back to later.

Consider examples, such as a bank deposit transaction or an HMO reimbursement transaction, that do all of the work involved from the user interface to the database update. If these were implemented as a self-contained entity that included the code for all the layers in a multi-tier environment (the user interface, web server interface, middleware access, and database access) we can think of them as Vertical Components. In other words, we can define a Vertical Component as an object that implements a business process. It is what the analysis phase of a project tells us we need for a specific business task.

A vertical component can be defined in terms of its inputs, its business rules, and its outputs. It is what an engineer would call a “black box” — a device that performs a specific function and does not require us to look “under the hood.” All we care about is what it does, not how it does it. When properly designed, vertical components can be inserted into any system and they will do the required processing. Better still, if the business rules change the vertical component can be modified without affecting any other part of the system.

When we build a system it consists of some number of vertical components. Note that at the level of the vertical component we don’t care about data access, or human interface, or deploying the application on multiple servers. These are all details that are hidden in the vertical component (don’t worry, though; we’ll come back to them later). Many IT departments are developing this way because they develop based on Functional Requirements (which generally define all aspects of a transaction) rather than a Technical Specifications. Technical Specifications typically include functional decomposition that permit the application to be developed “horizontally.” And Technical Specifications are absolutely essential when working in a pure Object-Oriented environment (which may explain the failure of so many OO projects). But the time to prepare a technical specification is a luxury that many shops cannot afford.

Looking at it from a different point of view, when shops take the time to first prepare Technical Specifications the work is frequently partitioned horizontally. As a horizontal example, when developing a multi-tier banking system a database programmer might write a service that validates an account number and another that credits a deposit to the database. A GUI programmer will create a screen that collects the account number and amount from the user and forwards that information (using middleware) to a service that contains the business rules for deposits. This service will then call the two database services, get the results, and return them to the GUI piece of the application. If it’s a web-based application a web programmer will translate the GUI data to HTML (or Java) to send to the user’s browser. Thus, for one transaction we potentially have four (or more) programmers involved and five interfaces to negotiate. We have developers who design the database services, developers who design the human interface, developers who manage the web server, and developers (who sometimes seem to be trained in black

---

magic) who manage the middleware. In other words, our development team structure is horizontal rather than vertical, so the end result is horizontal rather than vertical.

The difficulty with the horizontal development scenario is that it shifts the focus from the business problem to be solved to technical details, and frequently the business need suffers. Programmers tend to think of their own piece rather than the overall business objective. They worry about messaging middleware issues such as sockets, or remote procedure calls (RPCs), or object request brokers, or asynchronous messaging and queuing technology. All of these distract the programmer from the real job, which is focusing on business functionality.

Coming back to vertical components, the concept is compelling but the implementation is not yet clear. At first glance we have replaced one set of problems with another. True, all the business rules are in one object, but that object has seven or more application programming interfaces (APIs) within it, as shown in Figure 1. There is user interface code that uses one or more Web APIs. There is the interface between the web server and the application using CGI, ISAPI, or some other not-quite-standard. There is a middleware client API. And a middleware server API. If a mainframe is involved, there is a legacy system API. There is the relational database API. And finally there is the API to whatever development tool we are using. It looks like we require the application programmer to be familiar with all of these APIs. (Not to mention recovering from errors in all 7 levels.) Virtually all of the development tools that use code-generation implement vertical components.

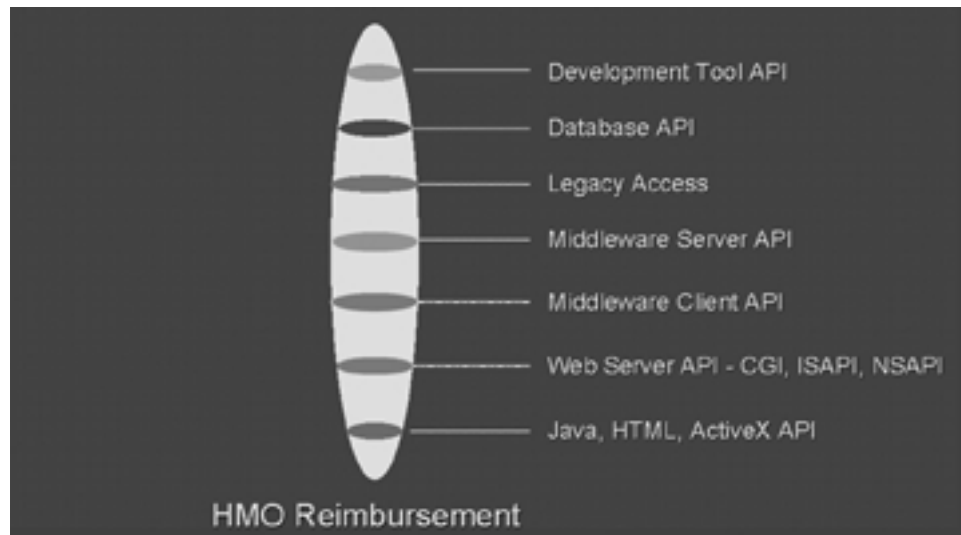


Figure 1

If we have multiple vertical components (and we will; there would be no point in using vertical components if there was only one) we note that EACH vertical component uses

---

the same seven APIs, and each must have code to deal with each of the APIs (Figure 2). If we want to change an API at some point in the future it must be changed in all vertical components. It appears that the vertical component concept by itself is not going to win any hearts and minds. There IS a solution, however, that combines the best of the horizontal and vertical approaches.

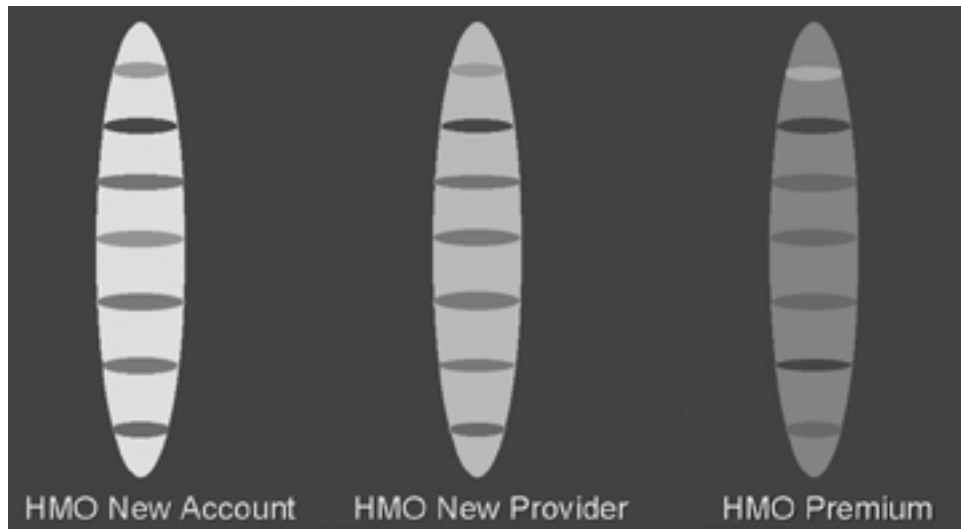


Figure 2

## Synergy

If we look again at several vertical components we note that each has pretty much the same API calls. We can leverage this fact by defining a Horizontal Component as one of those APIs across all vertical components, as shown in Figure 3. We thus have processing contained in the horizontal components and business requirements in the vertical components. But we still haven't built anything; all we have is some lines on paper!

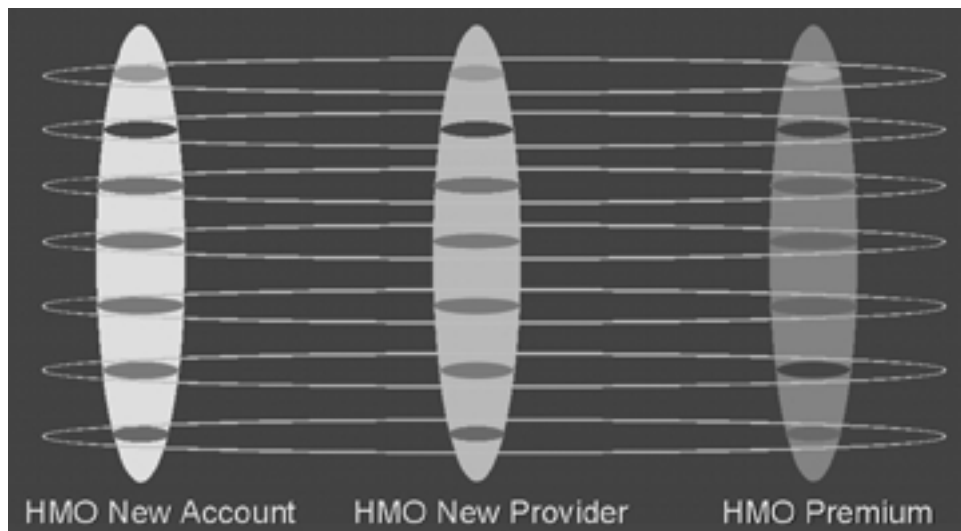


Figure 3

What we can now do is extract the horizontal components into their own space, as shown in Figure 4. We provide a consistent set of wrapper functions for the horizontal components, reducing all of the horizontal components to a single API. This removes, for example, the details of interprocess communication from the vertical components,

---

allowing the developers to concentrate on business logic instead of communications. We have essentially reduced the processing to one API. Of course, someone has to build this API. Fortunately, Prolifics has already done this for you; the Prolifics API essentially IS the only API you need to build multi-tier applications.

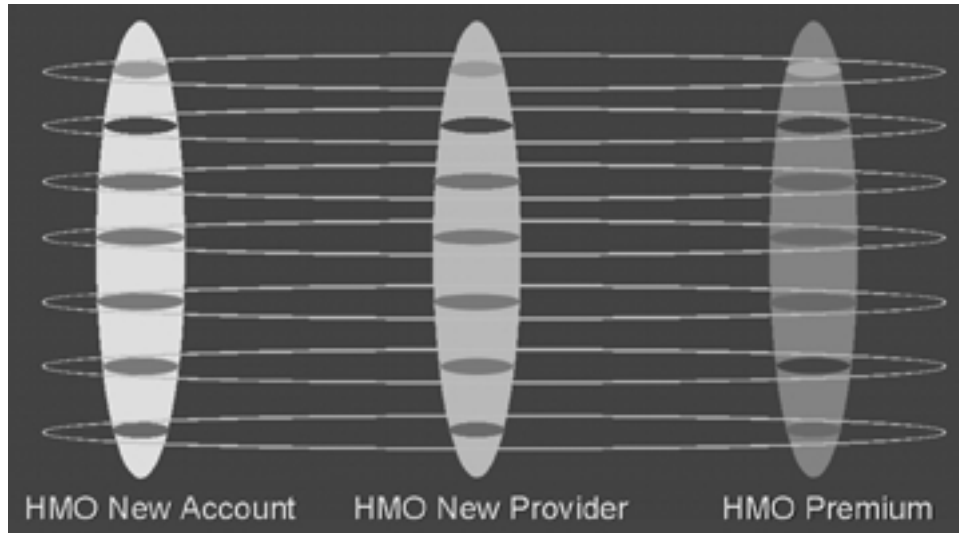


Figure 4

Horizontal components must be “intelligent” — that is, they must have a means of getting information about the vertical component that is using them and what the vertical component requires. Consider, for example, the database API horizontal component. If a 2-tier client/server developer wanted to access the database, he/she would normally write SQL (or have the development tool write SQL from QBE or a wizard, based on the fields on the screen). In a multi-tier environment, however, the actual database interface code is on the server, not the client where most client/server tools reside. We could still design the database horizontal component to accept an SQL statement as a parameter, but we have lost some of the advantage of our tool. Thus, 2-tier tools are not a good solution in a multi-tier world. Instead, we should allow the horizontal components to cooperate and exchange data with each other. (This is the concept behind Prolifics’ Transactional Object Model, which we will get to in a moment.)

Using horizontal components, the developer creates a screen with fields. The screen also contains information on the relationship between those fields and the tables whose columns they represent, as well as any foreign key information from the database schema (essentially, a query tree). These “metadata” (data about data) are obtained by Prolifics by querying the database and saving them in the Visual Object Repository. When the user requests an event (for example, retrieve data for viewing) the user interface horizontal component first protects all of the fields, then passes the field list, the query tree, and the user event (“select” in this example) to the database horizontal component.

The database horizontal component then dynamically constructs the SQL statement required and returns the results to the user interface horizontal component. If the user had chosen a different database event (e.g., “insert”), the actions of the application code would be identical: The invocation of a database access. The user interface horizontal component would then get values from all of the fields, passing them and the field list, the query tree, and the user event (“insert” in this example) to the database horizontal component. Thus, we have one piece of common code (the database horizontal component) that is used in every instance of database access. This same principle applies to the other horizontal components; we have essentially made all of the processing code

that is not business-rule specific reusable. Note that it is not easy perfecting the interface between horizontal and vertical components. But Prolifics has been at this for 20 years.

An important consideration is the ability to deal with unique or unusual situations, or factors that were not apparent at the time the horizontal components were created. To be truly robust the horizontal components must be extensible both globally and for specific transactions. For example, in a specific transaction you might want to modify the component-generated SQL, but leave the overall model intact.

## The Visual Object Repository & Transactional Object Model

We can apply the same abstraction to business rules and to the user interface. That is, remove the business rules to a repository and the widgets to a class library (or better still, the same repository). As Figure 5 shows, the vertical components inherit their processing from the horizontal components, their look and feel from the class library, and their business rules from the repository. In Prolifics the Visual Object Repository (VOR) captures the application's metadata, including database schema, business rules, widget customization, custom graphics — everything needed to build vertical components and provide information to the horizontal components. The entries in the VOR are an example of multiple inheritance — a concept from object-oriented technology that is rarely applied in practice. A VOR object inherits its look and feel from the widget library of the platform the application runs on, its metadata from the database schema, and its business rules from the developer.

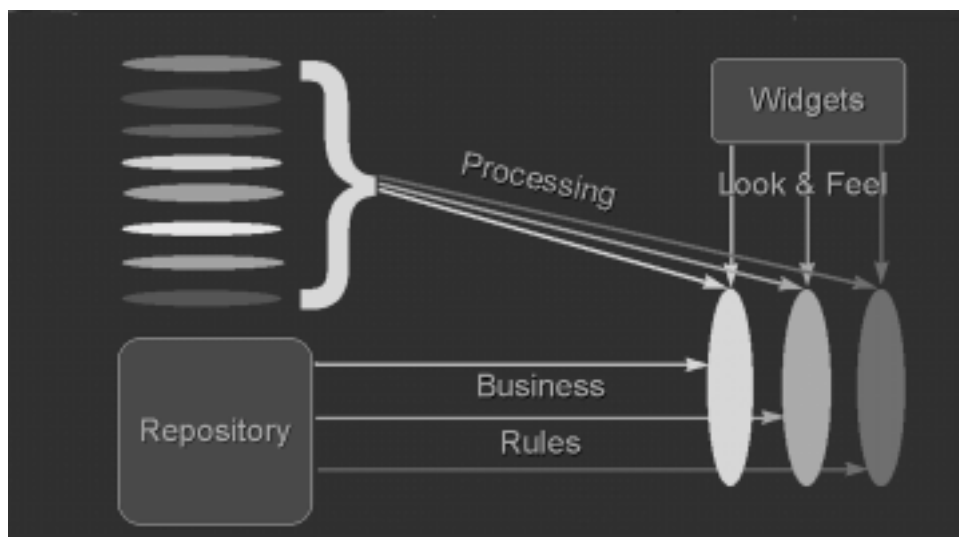


Figure 5

Prolifics applies another abstraction to the horizontal model. If we define a Processing Object as the set of horizontal components (Figure 6), the vertical components (or applications) then deal with only a single object for all processing. The horizontal

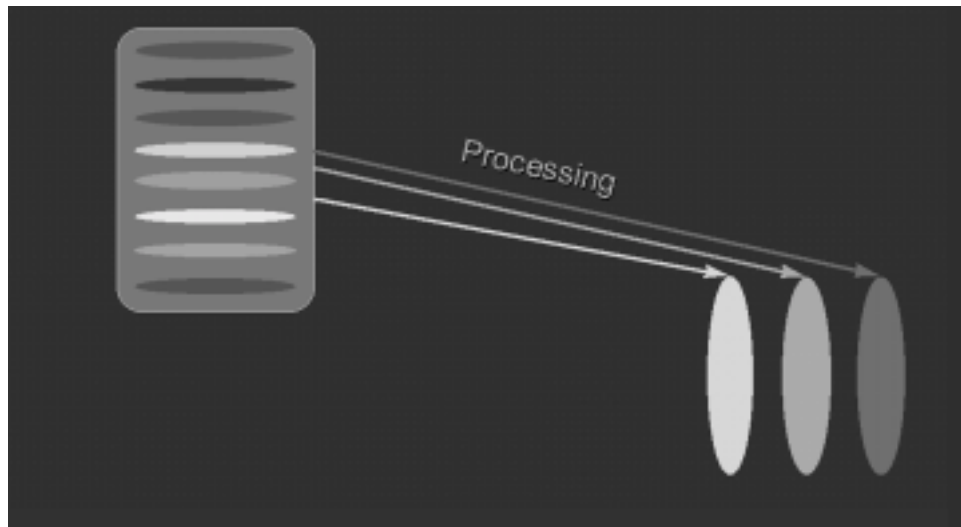


Figure 6

components within this processing object communicate with each other in addition to communicating with the vertical components. This permits transparent error processing and recovery as a side benefit. If properly designed, the processing object can be reused across projects and even across business units. This is reuse on a grand scale! In Prolifics this processing object is included in the development environment and is called the Transactional Object Model (TOM). The TOM is essentially the set of horizontal components encapsulated into a single processing object that is ready to use out-of-the-box, but that can be customized to meet individual project needs.

We get other benefits from the VOR and TOM. Since the objects in the VOR inherit their look and feel from the platform, the same application can run on different workstations, and will take on the appropriate user interface for that platform. Thus, the same application can be used on the World Wide Web (using browser controls), any flavor of Windows, Macintosh, Motif, or even character-mode terminals. The vertical components are unchanged across platforms.

Similarly, the TOM encapsulates the database interface, so applications can be independent of the specific database engine chosen, and, again, the vertical components are unchanged if the database engine changes or if multiple heterogeneous databases must be accessed within the same vertical component.

We can also extend the environment without necessarily modifying the vertical components. Suppose, for example, that we discover we need an added security layer such as data encryption. We can create a new horizontal component and add it to the TOM. We may have to modify the adjacent horizontal components, but the addition can usually be transparent to the vertical components that use the processing object. When the vertical components are affected (for example, if multiple security levels are added, requiring the user's privileges to be checked) the change can frequently be made by adding a property in the VOR that is then distributed to the vertical components.

A question that may come to mind is "what overhead is incurred in all of this sophisticated processing?" In practice, system performance is improved by a structured approach such as supported by Prolifics. The real reasons for poor performance are under or mis-utilized technology or poor design, not how tight the code is. And the Prolifics horizontal components themselves are written in C, so THEIR performance is not an issue.

## Developing Multi-tier in One Step

We have thus far glibly passed over the fact that multi-tier applications still require partitioning the application (or, in our examples, vertical components) into clients and servers. In the worst case we have web clients, GUI clients, web servers (which are “clients” to the business logic servers), business logic servers, and database servers. Thus, each vertical component appears to have several distinct pieces of code associated with it. With proper use of the VOR and TOM, however, we can reduce this plethora of servers and clients to only two: the user interface and the business logic. The VOR and TOM are smart enough to handle all of the other server functions transparently.

First, consider the user interface. A user interface for a specific vertical component has specific fields and specific events that it must recognize. These fields and events are the same regardless of what hardware platform users have on their desks; the only difference is the presentation. As we mentioned above, it is unnecessary to develop screens for a specific platform. Instead, Prolifics creates a data structure that defines the fields and events, then uses different user interface horizontal components to display those fields and events on the user’s display device. For a web browser the user interface horizontal component would draw a form with HTML, Java, or ActiveX dynamically from the form’s data structure. Your developers don’t even need to know HTML. You could even have different user interface components for different browsers, and further customize the display.

With the user interface out of the way we can now address the back-end (server) processes. Most tools let you down at this point; you are stuck writing the servers (which can be 80% of the whole application) in conventional 3GL or 4GL code. Prolifics solves this with Visual Server Development.

## Visual Server Development

Traditional multi-tier development requires you to create user screens with a client server tool or a class library and *services* written in C or C++ that run on the server. You then use a messaging mechanism or a remote procedure call from the user screen to the services you have created. There’s a lot of work involved in creating message buffers, sending them, waiting for a response, and processing the return when (or if) it comes. And if it doesn’t come there’s all the code required to determine what the error was and to recover from it. With Prolifics, however, you create the user screen and the service or services that support it at the same time with the same tool. In the time it takes to create a user screen you can also create a server process and the associated processing and communications.

The Prolifics development environment is both a sophisticated client screen tool and a service creation tool. Prolifics accomplishes this feat by using a visual metaphor for creating services. We call this metaphor the *service container*. A service container looks like a screen and is drawn like a screen, but at runtime it becomes the data structure that the server uses to determine the processing required within the service. What this means is that you can draw your service functions just as if they were screens. You can create a simple service container by just copying the client screen. You create more complex services the same way, but you can then embellish the service container with custom processing. Better still, using the Screen Wizard and Prolifics will create both the user screen and the service container in one step.

---

The “fields” in the Service Container define the input and output to the service. At runtime Prolifics queries a runtime metadata store to determine what these are. The application business rules that are embedded in the Service Container are inherited from the same Visual Object Repository as the fields on the client screen. Thus, if you change a requirement, both the screens and the Service Container are automatically updated at the same time. You can also partition validation between the client and the Service Container by manipulating the VOR.

## Transparent Middleware

The service that processes the business rules is all that is needed if the middleware and database horizontal components are “smart” enough. We have already shown how this can be done for the database component. The messaging middleware layers need some additional intelligence also to make them transparent. The trick here is to make a common data store available to the middleware horizontal components that describes the vertical components. This common data store can be accessed from both the client hardware and the server hardware. It lists the vertical components and the inputs and outputs of each. Thus, all a “client” within a vertical component has to do is call the processing object requesting service. In Prolifics, this structure is called the Interface File and is accessed through the middleware from both clients and servers. The client middleware horizontal component accesses Interface File, identifies the service and the arguments it needs, and invokes that service, extracting the required data from the client. The server middleware horizontal component accesses the same Interface File to determine what the characteristics are of the client that is invoking it. It thus knows what data to expect and what data to return.

Figures 7 through 12 illustrate the amount of code savings that can be achieved with effective use of vertical and horizontal components. Figure 7 shows middleware API code that would be required to process a bank deposit transaction. The API used is BEA’s Tuxedo. We assume that the client screen has already been validated when this code executes. The *get\_field* functions retrieve the values of fields the user has entered. Figure 8 shows the corresponding service that processes this request. It gets its parameters using the Tuxedo middleware API, and it calls directly to the database server, using the database server’s API.

Figure 9 shows the same client code when used with an intelligent middleware API horizontal component. `SERVICE_CALL` is the generic processing object API that invokes a service on a server.

Figure 10 shows the service that was invoked. It calls the database API directly. If the “intelligent” database horizontal component were to be included, the resulting service could be implemented as shown in Figure 11.

## Who Needs Code?

The answer is “everyone.” The true 4GL tool that creates complete applications without writing any code just doesn’t exist. Some tool makers pretend that they can do codeless development; however, the result is frequently a product that makes it very difficult to add code if necessary. This is especially true of code-generation products. If you generate the code with the tool and then modify it, what limitations does this place on future editing with the tool?

---

The Prolifics approach is to do as much as possible without code, but to let the developer decide when and where custom-written code is appropriate. And, since Prolifics generates data structures rather than code, there are no limitations on future changes to the application that take full advantage of the Prolifics development environment.

## **Extending the Model**

If Prolifics only provided a powerful tool for building applications the way Prolifics thinks you should build it, you could reasonably ask the question “what if I have special requirements that Prolifics didn’t think of?”

Well, Prolifics thought of that also. Just about any function provided by the Prolifics development tool, horizontal components, or run-time library can be extended with functions that you write. Prolifics is a “base” system that can be extended in almost infinite ways, such as adding your own functions to the API, changing the way the TOM parses your forms, changing the way the VOR imports from the database, or managing regions of the screen at a pixel level.

Do you want to see what your users are typing on a character-by-character basis? Just add a “hook” function to the Prolifics input function.

Did you see (or develop) a powerful ActiveX component? Prolifics will integrate it seamlessly into your Prolifics application.

Do you have hotshot web designers who want to create their own pages? That’s no problem either. Prolifics, through its ProActive Web features, permits you to create your own pages but still interface them to Prolifics servers on the back end.

In, short, you can modify almost any function of Prolifics to meet your unique requirements.

## **Putting it all Together**

The current state-of-the-art in multi-tier software development is not achieving the gains that were expected of it. The reasons for this can be debated, but the bottom line is that something must be done if timely and cost-effective enterprise-quality systems are to be developed.

We have shown that one way of getting significant improvements in new software development productivity is a hybrid approach using both horizontal and vertical components, allowing the vertical components to inherit from multiple components. (While this is not the strict definition of multiple inheritance in objected-oriented development methodologies, it is a similar concept.) The key is to build intelligent horizontal components that query data structures in the application’s vertical (business oriented) components. Prolifics is the only tool that does all of this for you.

---

```

#include <stdio.h>          /* UNIX */
#include <string.h>        /* UNIX */
#include <fml.h>           /* TUXEDO */
#include <atmi.h>          /* TUXEDO */
#include <userlog.h>       /* TUXEDO */

int client_deposit()
{
    char *error_message;
    char *receive_buffer;
    char *send_buffer;
    char *account;
    char *amount;

    /* Begin transaction */

    if (tpbegin( 10, 0 ) < 0)
    {
        print_monitor_error( "tpbegin" );
        return( -1 );
    }

    /* Allocate and load send data buffer */

    send_buffer = tmalloc( "FML", NULL, 1024 );
    receive_buffer = tmalloc( "FML", NULL, 1024 );
    if ((send_buffer == NULL) ||
        (receive_buffer == NULL))
    {
        print_monitor_error( "tpalloc" );
        return( -1 );
    }
#include <stdio.h>          /* UNIX */
#include <string.h>        /* UNIX */
#include <fml.h>           /* TUXEDO */
#include <atmi.h>          /* TUXEDO */
#include <userlog.h>       /* TUXEDO */

int client_deposit()
{

```

```

/* Execute the call */

if (tpcall( "DEPOSIT",
            send_buffer,
            0,
            &receive_buffer,
            &receive_length, 0 ) == -1)
{
    if (tperrno != TPESVCFAIL)
    {
        tpabort( 0L );
        print_monitor_error( "tpcall" );
        return( -1 );
    }
    tpabort( 0L );
    error_message =
        Fvals( (FBFR *) receive_buffer,
              MESSAGE,
              0 );
    if (error_message == NULL)
    {
        print_error(
            "Invalid FML field: MESSAGE" );
        return( -1 );
    }
    msg( error_message );
    return( -1 );
}

/* Commit the transaction */

if (tpcommit( 0 ) < 0)
{
    print_monitor_error( "tpcommit" );
    return( -1 );
}
return( 0 );
}

```

```

#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <fml.h>           /* TUXEDO */
#include <atmi.h>          /* TUXEDO */
#include <userlog.h>       /* TUXEDO */
#include <deposit.h>       /* FML fields */

EXEC SQL INCLUDE sqlca;

void DEPOSIT( transb )
TPSVCINFO *transb;
{
char amount_buffer[20];
char *return_buffer;
int status;
EXEC SQL BEGIN DECLARE SECTION;
double amount;
char *account;
EXEC SQL END DECLARE SECTION;

/* Allocate return buffer */

return_buffer = tmalloc( "FML", NULL, 1024
);
if (return_buffer == NULL)
{
userlog( "Memory allocation failure" );
treturn( TPEXIT, 0, NULL, 0L, 0 );
}

/* Unload input buffer */

status = Fget( (FBFR *) transb->data,
AMOUNT,
0,
amount_buffer,
0 );
if (status == -1)
{
userlog( "Invalid FML field: AMOUNT" );
treturn( TPEXIT, 0, NULL, 0L, 0 );
}
}

```

```

EXEC SQL UPDATE ACCOUNT
SET AMOUNT = AMOUNT + :amount
WHERE ACCOUNT = :account;
if (sqlca.sqlcode != 0)
{
sqlca.sqlcode = 0;
status = Fadds( (FBFR *) return_buffer,
MESSAGE,
"Account update failed."
);
if (status == -1)
{
userlog( "Invalid FML field: MESSAGE" );
treturn( TPEXIT, 0, NULL, 0L, 0 );
}
treturn( TPFAIL,
0,
return_buffer,
0L,
0 );
}
if (sqlca.sqlerrd[2] == 0)
{
status = Fadds( (FBFR *) return_buffer,
MESSAGE,
"Invalid account number."
);
if (status == -1)
{
userlog( "Invalid FML field: MESSAGE" );
treturn( TPEXIT, 0, NULL, 0L, 0 );
}
treturn( TPFAIL, 0,
return_buffer, 0L, 0 );
}

/* Load return buffer and return */

Fadds( (FBFR *) return_buffer,
MESSAGE,
"Update successful." );
if (status == -1)
{

```

