# RECORD, Volume 23, No. 3*

Washington Annual Meeting
October 26–29, 1997


## Session 4PD
## Developing to Distributed N-Tier Client/Server Applications

**Track:** Computer Science
**Key words:** Computer Systems


**Moderator:** MICHAEL F. DAVLIN
**Panelists:** J. PETER DONLON†
JONATHAN ZUCK‡
**Recorder**: MICHAEL F. DAVLIN



*Summary: Current software development methodologies view corporate applications as comprising at least three tiers. The first is a client tier, which provides graphical user interface (GUI) and other services to the end-user at the desktop. The second is a database tier, which provides data management services to one or more applications or processes. The third is a business rules tier, which provides centralized services to applications related to the rules and characteristics pertaining to the company, its operations, and its products. Other tiers, such as high-performance computational servers may also be present in n-tier architectures. Historically, actuarial applications have blurred the distinction between these logical components that are present in all applications.*


**Mr. Michael F. Davlin:** The first speaker is Jonathan Zuck, who is the vice president of technology with Financial Dynamics in McLean, Virginia. Jonathan's a popular speaker at a variety of professional conferences largely targeted at Microsoft technologies and Visual Basic. He has written for a variety of magazines like *PC Week* and *Visual Basic Programmers Journal*. He's going to talk about the technology and the theory behind n-tier distributed development. Then, just to convince you that we're not building castles in the air and trying to live in them as well, we have Peter Donlon here to talk about USA Administration Systems'



†Mr. Donlon, not a member of the sponsoring organizations, is Senior Vice President, Marketing and Sales for USA Administration Services, Inc. in Charlotte, NC.

‡Mr. Zuck, not a member of the sponsoring organizations, is Vice President of Technology at Financial Dynamics in McLean, VA.

experience in developing an n-tier client/server application for their new product line.

We're going to start with Jonathan and talk about the theoretical aspects of this, and give you a demo of some of the technology that's available. Then we'll follow up with Peter.

**Mr. Jonathan Zuck:** I'm here to talk to you about three-tier client/server applications in general, the concepts behind them, how they marry the aspects of object-oriented programming and real transactional programming that we've left behind and we're sort of revisiting. I'm going to talk a little bit about some of the theory and the design concepts associated with three-tier programming. Finally, I'll talk a little bit about some of the technology that's emerging to facilitate this type of programming because the principle isn't really new. Only the facilitating technology that makes it a more straightforward operation to develop an application in this fashion is fairly new.

I've given many speeches to CEOs and boards of directors about this material at a somewhat higher level. I'm going to talk about client/server application development in general, make a distinction between a logical and a physical architecture for application development, and provide some information on more traditional architectures for client/server and computer development. Finally, I'll talk about application partitioning, which is this idea in the PC world of distributed computing.

So what is client/server? Client/server gets used in many different contexts by many different people, so I want to get my definition out on the table. When I talk about client/server, I'm talking about distributed processing, which means one computer program talking to another computer program together synergistically in order to produce a result. So if you have a classic database application running on multiple machines all sharing a file on a file server, that would not be client/server. Instead, when you're talking to some kind of application on a server, that's going to be a client/server application. Some responsibility is maintained by the server side application itself.

A classic example is the library. If you go to your local public library, you use the Dewey Decimal System. You check through the card catalog, find the books that you want, go to the shelf, and pull out some books. You take the books to the table, decide which ones to check out, and then you leave the library. That kind of a system gives you a lot of control. You go right to the information in the library, and there are also some efficiencies associated with that. I can see things that are nearby, and I can choose very quickly what I want, and I can get out of the library

in a very efficient way. But some of the problems associated with that had to do with integrity and efficiency. Those books I left on the table aren't in the stacks, but they're not checked out, so somebody who is looking in the card catalog will think that they're there but they're not. I've done some things to disrupt the integrity of the library. I might reshelve it in the wrong place, which is even worse than leaving the books on the table. The other problem is one of efficiency. If the library discovers that 10% of the books are the ones most often checked out, it might move those books to the front of the stacks so that people can get to them faster and get out of the library more quickly.

It seems like a more efficient system in a limited sense. In a broader sense, it's a less efficient one. We can go to the Library of Congress or the New York Public Library that has closed stacks. I can go up to a librarian with a white slip of paper and say, "I want this book." She disappears for a while or hands off the slip of paper and comes back later with the book. That facilitates a much more efficient system underneath. At the Library of Congress there are something like six or seven separate libraries and tunnels and escalators to get to different stacks. Being able to go to any one of those libraries, giving a request to the front desk, and then not having to worry about where it's coming from or how it gets back to where it needs to go is what makes that distinction fundamental; the same thing is true in client/server. So when we start talking about client/server, what we're trying to do is have not only the reader but the librarian in place. As we start to develop and modernize the concept of client/server, we want to keep that fundamental concept in mind because that's what's really going to lead to its greatest efficiency.

When we talk about a logical or a physical architecture in client/server, we mean two different things. In a logical architecture, you're defining the abstractions. When I'm talking about the reader and the librarian, I'm talking in general terms about the interactions of different entities within the system. So there's a real emphasis on the conceptual design of the system rather than the physical implementation. As far as the physical architecture is concerned, we are starting to talk about the implementation details, the languages that will be used, and how things might get partitioned across the screens, and so on. So I'm beyond thinking about logical entities and I'm beginning to instead think about physical components that are going to make up the distributed system.

In a traditional logical architecture, the user interface, or the thing that you actually view your application through, actually has a good deal of intelligence in it. Just as the reader goes to a library with a lot of knowledge about the Dewey Decimal System and how to use the internal functions of the library, the same thing applies to the user interface in your application. It has to not only be business rule aware, but also aware of the design of the underlying database. Again like the design of

the underlying library—that traditional architecture means the changes to that design are going to require significant re-education, or in the case of application, a redesign of that client side application. So I have kind of a flat model integration with proprietary connections where I end up accessing the data in the way that makes the most sense to me for my application, which only leads to greater complexities when other people try to share that resource.

Let me talk about some traditional physical architectures, just so we can make this distinction clearly. There's a centralized architecture where you have kind of a smart back end and kind of a dumb client. A file server architecture is the traditional database-type application that really exploded in the PC marketplace because of the inefficiencies associated with centralized programming. We did flat-model reorganizations through business process reengineering. We told departments of ten people, "You are responsible for your own set of goals and your own productivity. Go for it!" The only way that could happen is if those departments could also have some autonomy from the standpoint of information technology (IT), and that's why we had this explosion of these file-server-oriented, simple database-type applications. As we started to see the inefficiencies associated with those systems, because we've built in stove pipes that couldn't talk to each other, we started to realize that we have to share the database. The best way to do that is through traditional client/server applications.

So in a centralized architecture, you had a terminal sending key strokes to a mainframe-based application that talked to its own database, and then you basically sent screens or characters back to a VT-100 terminal or something like that. In a file-server architecture, the basic physical architecture has an application sitting on the workstation that is using disk I/Os to directly access the disk of a file server that was then serving up the data. Of course, there's some problems associated with that. If I have a shared D-base file, then things like referential integrity start to get a little bit suspect. If I have a D-base file, and the first byte of that record is an asterisk, that's a deleted record. As a result, I can go in through any kind of tool and change that and disrupt the references that we have in a fully normalized database.

In a two-tier client/server application, which is really what's been going on the past five years with tools like Tower Builder and Inside Visual Basic and Object View and some of the other tools, I have a workstation application that uses a generalized communication. This is like the white slip I give to the librarian, which is SQL. SQL stands for structured query language. It is the way I communicate which "book" I want. I have an engine running on the server side that goes about getting that information and returning result sets or data sets back to the workstation.

The newer concept here is the so-called three-tier architecture or three-tier client/server. It has also been called an n-tier (because it could have more than three tiers), a multi-tier, or Microsoft's term, the services-based architecture. If we think about it in three different abstract layers, we see we have user services, business services, and data services. The user services are the applications that the user is working with to either input or collect data. That's what we're going to refer to. Like the dashboard of your car, this is the way that I interact with the application.

Business services actually contain the core business processes and rules that make up the actual functionality of that application, which is often going to be a very simple functionality. It harkens back to some of these old mainframe applications where you had to enter data in a very specific way and push “Go” and you were done. The idea behind the separation is that these user services can be more tailored to the way the users want to collect and analyze data, but the underlying business processes that are being exposed are still very straightforward executions of transactions much like the ones that we were used to in the mainframe world. The data services are typically for referring to an SQL-based database or a relational database engine sometimes with some additional layers on top of it in the form of stored procedures or triggers, or even, in some cases, components.

When I try to design a three-tier system at my firm, we try to err initially on the side of too many layers. We try to think in logical terms about all the different entities that make up this application. So in the user services layer, we often take the user interface and separate if from the application itself. This is similar to Microsoft Excel. There is an interface that you work with, if you're working with spreadsheets, but I can also programmatically modify, create, combine, and filter spreadsheets from another application through an application interface (API) into itself so there's a separation there.

In the business services, there's typically the separation between business processes and business rules. As we start to design more complex systems, we want to be as granular as we need to be to accommodate change. So if some particular business rule is the thing that's most likely to change over time or with greatest frequency, that's the thing that we're going to want to isolate so that it's more easily replaced down the road.

Regarding the data services, you might have database specific services in the form of procedures that are specific to that database as opposed to generalized services provided by the relational incident itself.

Let's think about this model in practice. If I take a large order entry system, my first step is probably going to be to break that out into a series of components that encapsulate the core business processes that I'm trying to reveal to the business environment as a whole. In this case, I have a products component, an inventory component, an order component, and a customer component. Those are independent units of work or business processes or sets of business processes that are not only available to the order entry system, but can be made available to a future system as well. For example, instead of the sales information systems having to start from the ground up or learn about the underlying data structures used by the order entry system, it is instead getting to its information via the standard components so that those contracts remain static.

Changes to the database as well as changes to say the pricing rule can be subject to change without having an impact on the user interfaces that have been built. We've thereby lowered the overhead associated with creating multiple user interfaces for a multiple group, and this is a key component to productivity among the user. When we talk about bringing the computer to the user instead of bringing the user to the computer, it's not a matter of leading horses to water; instead it is fundamental productivity in which data collection and data analysis reflect business practices. Our job is to facilitate business practice, not to dictate it. In order to do that, we take the core practices of that business and make them separate from the way in which they're used.

One of the benefits of the service model is that it allows us to make more flexible solutions. I can create multiple interfaces, and I can change things about the application. It's also a more efficient use of skills. Frankly, some of us are better at creating user interfaces and some of us aren't. Some of us are better at creating efficient codes to the execution of transactions to facilitate scalability of the system. That's a very real skill, but the person with that skill is probably not the same person who's going to go through and duke it out with the user for the ideal user interface. To them, the ideal user interface looks like a mainframe. From the standpoint of scalability, that is the ideal user interface.

So it's the marriage of those two that involve specialization. There are opportunities for parallelism. Instead of this huge, sequential development that is taking place, I have interaction going on with the users for user interface, whereas the core business processes and rules are being developed by another group with another potential group of users and another group of developers. It facilitates interoperable technology. I might find a better way to do something. For example, my order entry system uses one credit processor and I want to change credit processors, so I can make that switch to a single component instead of digging into the whole system. It facilitates the distribution of data and processes.

As soon as I've made this logical leap component, then moving those around to different machines to increase the overall efficiency of the system requires a much smaller effort than it would when I'm trying to undo a huge monolithic Powerbuilder application that I created five years ago. It's amazing that we're in a market right now where Powerbuilder applications are considered legacy applications, but that's what's taking place. It's this huge wreck of an application where change means digging into data windows and digging into things that I did to try and optimize the use of that application on a 486-66. I would rather take pieces of that and distribute it out across multiple servers and multiple machines and multiple departments in order to increase the efficiency of the organization as a whole.

The final benefit of the service model is it encourages the reuse of components. This has been a dream for some time. We talked about object-oriented programming not really working, and this is the greatest new fad. I'm one of the most cynical people in the world. I don't think anything is the greatest new fad which is probably why I'm still talking about this instead of `Java. By taking components and developing kind of a contract as to how they will be used, it allows me to change how they perform the operations they're designed to perform without really changing the way in which they're used. That facilitates use and reuse across multiple applications.

Let’s discuss the implementation details of the three-tier client/server—the so-called physical architectures associated with it. First, it is traditionally done with code models. Many of you who are developers have been trying to implement things of this sort without the technology being available by trying to encapsulate functionality with the tools that were provided. It's often done with code models so there has been a movement toward more object orientation as those tools have matured, which gives me this kind of encapsulation of both information and process. In coder terms, we say code and data. We're trying to take the process that we want implemented and the data required for that process to take place and encapsulate them together. Instead of having to have the data understood by everyone, it only needs to be understood by the processes that need to access it, thereby eliminating the need for things like database change requests and huge documents of that sort that have really plagued the productivity of such of an application.

One of our clients is a government insurance company that was $10 million into a project to automate the business processes before a line of code had been written. Three years of work generated binders of paper to throw over the wall to the next group, but nothing had been accomplished. That is because the overhead associated with getting one little thing done is too great. There is such a large

number of people involved in getting that one little thing accomplished. This modulated development is part of what facilitates the parallelism necessary to move the whole beast forward.

Some physical architectures have included a single application, a partitioned application, or a distributed application. So again the marriage of logical to physical doesn't automatically mean just because I'm building some small departmental application that I shouldn't be thinking in these terms. I can still have a three-tier application that isn't client/server in the classic sense of being distributed across machines.

Finally, during the implementation of three tier, we tried for some language independence, and even that is a fairly recent development because of COM or the common object model.

We think of a single application architecture as an encapsulation of process and information that is taken in the form of modules, classes, or C++, original Basic, or even custom control.

In a partitioned application, we often think of dynamic data exchange as the means of two different components communicating although that's kind of a dying means across process communication, or implementation of business processes and rules in the form of dynamic link libraries. They can also be written in a compiled form that is easily distributed, easily updated, and is without change in user interfaces. A more recent one is object linking and embedding (OLE) automation. I use the term OLE automation even though that's an old term because it's the one that I think rings the most bells. When I say ACTIVEX, it means too many things. The bottom line is we're talking about something that is being referred to as COM or the common object model, and that is the means by which we published the sort of yellow pages of the business services that we create. We publish these services in a way that's easy to use across multiple languages.

In a distributed application, I could use network DDE although I think the remaining application of that is HEART. There's a remote procedure call so that I can use the very low-level method that has been around for some time. Those who are less timid among us have actually created some distributed systems in this way, but it's a very proprietary set of connections, and it's difficult for others to share the business processes that we've created. Mail application programming interface (MAPI) is something that has grown in popularity because of its bulk tolerance and the ability to do distributed processing through something other than a tight connection. So MAPI and Exchange etc. are emerging and it's probably going to emerge in the not-too-distant future with COM or automation or distributed component object model

(DCOM), which is really the means through which most component-based architecture is being implemented today.

I don't know if that's the architecture you'll be demonstrating, but DCOM is definitely the architecture of the future, at least the future according to Microsoft.

When distributing the services, there's a number of different options for how I might distribute these components. One is the so-called “fat client” or a GUI graphical user interface, the interface logic and the business logic all here on the client machine talking to the data access logic and the DBMS on the back end. This is what I would refer to as the Powerbuilder application development model. Then there is the “thin client,” using a fat server where I just have interface logic on the client and business logic and data access and the data on the back end. This is where people who were trying to mature from this first architecture were moving by taking things like fundamental business processes and implementing them as extended store procedures, etc. By building those things into the database, they ended up creating a reverse architecture. What we're really talking about here is a multi-tier architecture in which I have the GUI and the user services tied together. Then business services and some data services are encapsulated independently of the user services and the data services, and then data services are on a separate machine.

This final architecture is one that's getting a lot of attention. It is the multi-tier, Internet/ Intranet architecture. This is where the vocabulary is getting a little bit mangled because a lot of people now think client/server means your browser and your ACTP server, your internet information server (IIS) or your Netscape server. I think of it as a natural extension of the one before it. I still implement the two bottom tiers in the same way that I would have before with the database, business services, and so on being separate. However, I then combine the browser and the ACTP server to be one additional client or consumer of those business services. So I view your Internet server and your browser to be together as a client of those underlying business services. I don't see them as being a huge change or deviation from this architecture, but simply a new technology for the deployment of user services. When I'm in my IIS box creating ASP pages or something of that sort, what I'm doing is generating user services codes; I'm not implementing my core business processes in that fashion.

Development tools are basically any COM capable language, such as Visual Basic, C++ Delphi and Java. These are all languages that are capable of being a part of a component or partition system.

So if you think of this modern architecture where I have clients, business rules, and corporate databases, to some extent I begin to increase the complexity of the application that I'm developing. On one hand, it sort of increases its simplicity through the encapsulation of core processes and rules, making it easier to maintain and modify down the road. I have increased the initial complexity—its design complexities, its distribution complexities, and so on. I've also increased the complexity associated with making sure the system maintains the overall business integrity of the system because of things going on in different places. So we have this sort of new class of shared application codes that run from these middle-tier machines. This creates new challenges for component developers. There are more things to administer and understand.

There is new technology from Microsoft called Microsoft Transaction Server (MTS), and there are other CTS monitors, such as one that is available from Sybase. These products are designed to help provide the kind of administrative interface over these component-based applications. So MTS in this case will have the business rules and help manage the database. It makes it easier to create high-performance scalable and robust applications because it's going to simplify what we need to do to create them. Enterprise and Internet server applications use visual tools like Visual Basic.

MTS is a product for running business applications on the NT server. It's a kind of component manager. The same applications run from small work groups to large enterprises. That's kind of a marketing blurb right now. The idea is that we want to be able to create applications in the same way and then have them scale across different sizes of implementation. That's what MTS is designed to facilitate. Is it all the way there yet? No. Large departmental applications are going to probably be at the high end of the scalability right now, but the model is in place for scalability in the future.

MTS provides ease and flexibility of the PC for enterprise applications and reliability and scalability of mainframe solutions for PC servers. There are a bunch of mainframers out there going, “Hey, remember us?” There's a reason we put you through all that pain. The fundamental processes and scalability that was necessary and the integrity that was necessary for the system was something that was very important to that group. The pure transactional integrity was a big issue that we’ve somehow lost sight of in this “me generation” of PC-based application development. We're kind of making a transition back, but hopefully it won’t be all the way back.

There might be a series of components, such as inventory components, sales components, and a tax-calculation-rule-based component. They're all ActiveX components. When I talk about old-age servers, COM servers, or ActiveX servers, those are all the same things. ActiveX gets used in a lot of different contexts.

There's the ActiveX server architecture, and there are ActiveX controls, etc. Microsoft believes in trying to call them all the same thing because they're all based on the same underlying architecture called COM. But when I talk about these servers, I'm talking about noninterfaces. They're little components that are used only by other programs, and those are all managed by the Microsoft transaction server that runs under NT. I can get to it in a couple of different ways—the first is DCOM. I can have a client-based application that uses DCOM to get to my NTS component. The other is I can use ACPP from a client to something like IIS, which in turn programmatically uses COM to get to those components. I can sort of expend the availability of these components after the intranet or internet via the combination of COM and ACPP or I can just go to typical local area network (LAN) clients via DCOM. So I can have both thin and smart clients in terms of accessing the business office.

On the back end, what could those components get to in terms of data? This includes OLE transactions, databases, which right now there's only one, people server, but with MTS 20, there's support for XA, which opens up the world of Oracle and Form-X, Sybase, and so on. There's also a product called Cedar, which through SNA servers are going to facilitate access to CICS-type transactions via Microsoft transaction server. In other words, I'm going to be able to use COM-based interfaces to top the legacy transaction.

I create very simple, single-user components that run multiple users on the system. Thread and processes management is managed by the Microsoft transaction server. There is process isolation where, if I have a big Internet application, I have somebody coming from France and somebody coming from Germany. I have components that are maintaining information about exchange rates or pricing across different countries, obviously in a component-based solution. I don't want those to get mixed up somewhere in the middle of the process. So keeping new processes isolated is what facilitates multiple types of interfaces.

There is also resource pooling. Things like database connections are pooled so that they can be more efficiently used by the system. There is also automatic security, automatic transactions, and point-and-click administration. Now when Visual Basic 4 came out, it had something called remote automation. That was the beginning or the hint of this technology that allowed us to create these two-tier applications. It turned out that creating the plumbing associated with that application was the majority of the work because creating my little credit card processing server took a similar amount of effort as is needed to create something like SQL server because of all of the infrastructure associated with creating a good server. MTS is designed to bridge that gap.

Let's take a simple application like a transfer of money from one account to another. In a component-based application, I might design a component whose purpose is to withdraw and deposit money into a bank account or to manage a bank account, so that component could be used by itself to make deposits and make withdrawals. The idea behind reuse is that when I then want to create a transfer component, instead of building intimacy with bank transactions into that transfer component, it simply avails itself of the services of two account components. So a transfer basically means a credit to one account and a debit from another by accessing these two account components. In a distributed componentized architecture, there are more possibilities for failure. Failures are bad. In a world of financial transactions, this can be a serious issue. Applications must deal with many possible outcomes, so when we think about how the credit worked and the debit didn't, there are some issues there. If the credit succeeds, and the debit doesn't, money is created, which is good on one hand, and not good on another. There are some people behind bars now that thought that way. That really is a componentized architecture in which part of something failed. They're not aware of each other, so these components don't realize that they're part of the same transaction; they just know how to make deposits and withdrawals.

Part of our problem here is how do we keep this as part of one unit transaction so that we can have a single outcome? One possible solution is the transaction process monitor, a CP monitor, such as MTS. I round the whole thing inside of a transaction run by MTS, and the transaction changes to state all or nothing; either everything succeeds or nothing succeeds. They simplified the applications. If I know that these components don't need to be aware of each other and that the system, as a whole, would continue to function properly, that's going to simplify my life considerably. With no transactions, there are many outcomes, and with transactions, there are only two possible outcomes: success and failure of the operation as a whole.

A second challenge associated with creating component-based solutions, or really with creating solutions in general, is one of scalability. Building the internals or plumbing of a good server requires many different features. There's a receiver that's receiving requests for information across the network. There's a queuing mechanism to use up those requests so that I don't have deadlock and things of that sort. I have to manage connections and context so that I understand that two different users are getting two different types of information or the same information in two different ways. I have security issues to deal with, and I have threading issues to deal with. So these are all different pieces of the application that they need to build. There is also service logic, which is just the logic that I obtained from long meetings with the user about what kind of amortization schedules are going to be necessary in order to process a loan. Then I must deal with shared data.

I have to manage my components and I have to handle configuration of components, etc. A problem that many of us were facing was trying to somehow build these distributed applications with the raw technology of remote automation and DCOM.

An issue is the transaction server. It's designed to provide a lot of the plumbing so that all you're dealing with is the specific application. If you think about it, it's housing a bunch of little components for which Microsoft has chosen marbles to represent, and to script, I can get to them from Microsoft's Internet Information Server, they can get to back-office-type applications and information stores like Microsoft SQL Server and Exchange server, and they can also avail themselves of systems services like Active Directory, etc. And the transaction server provides the receiver that handles the incoming information and the queuing of requests from external applications. The connection management goes to the database and the connections to the components' scalability context so that again two users have process isolation, security, and rule-based security. Then managers can get to certain components in certain ways and clerks can get to others, like thread pooling, which facilitates the scalability on the back end without you having to worry about it, synchronization, and then transactions across the whole thing as well as a management interface for managing the components on the distant servers on which you distributed them.

A third problem is the complexity increases with scale. In other words, I have these different scales of applications—the desktop, the word proof department, the division, the enterprise, the Internet. So I have a huge number of users. As soon as I get to two users, I've increased my complexity considerably. This is something we face already. In shared data, I need to do optimistic versus pessimistic blocking. These things are still debated in terms of what is the best way to handle them. As they move up to work group, I also have connection management and security issues to deal with and a context to deal with. All these things come into play as soon as I go over one user.

As soon as I get past the work group and I'm into hundreds of users, I have to deal with multi-threading issues. Multi-threading takes full advantage of these advanced operating systems that we've been convinced to buy by making use of the threading capability on the server and managing multi-processing a little bit better. When I get to the division, I have to start doing load balancing, which means I've gone beyond the capacity of a single server. I need to start putting components on multiple servers, and then somehow dynamically decide which server to go to for them.

Finally, there is the enterprise. I have message queuing because I don't know how many enterprises are all sitting in one building on one network. Instead I've got some wacky LAN that's up some of the time or sometimes it is a 56K, and other times it's 128K. There's a lot of unpredictability. There's even organizations out there trying to use the Internet as their LAN. So we're going to start needing fault tolerance and things like queuing a message so that things will take place in a detailed fashion, if necessary. Finally, with the Internet, we need high availability. When I'm going to have hundreds of thousands of users, I'm going to need to have a capacity beyond anything that I've had to deal with in the past. In all honesty, the transaction server right now is probably somewhere in the middle of the division category or something like that in terms of this real built-in scalability. If you need to go beyond that, you're going to have to do much of your customized plumbing. The idea is that the underlying architecture behind MTS, from the standpoint of your component development, doesn't need to change; only some of the internals of MTS need improvement and growth in order to gain that additional scalability. The idea here is that one application can fit all. It's a growing ideal.

Finally, another problem is that precious resources limit scalability. When a client needs the server, resources must exist otherwise you end up in a lock-up state where you're waiting for resources like database connections to become available, scarce server resources limit scalability, the processes thread, connections, memory, etc. How to effectively manage those resources is another aspect of MTS. So components that are making use of these core business components, like the inventory component, say when they're done. They recycle resources such as processes, threads, memory, database connections, network sessions, component instances, etc. These are all things that MTS provides so that, ultimately, I can have multiple applications on multiple threads going out and sharing multiple components that are in turn sharing even more finite resources, which is the database connection. So it sort of builds down like roots of a tree if that makes sense. I can have an expanding availability associated with the resources through their reuse.

There is something we manage called packages, which basically represent the applications that I'm managing under MTS.

MTS will work in a more complex architecture as well. I think of a rent-a-car company with three main types of users. There are managers who manage inventory and price, there are clerks who are at the desk taking orders as people walk up to pick up a car, and then there are users on the Internet who do their own reservations. This is what I'm talking about in terms of the low overhead associated with creating multiple interfaces for different types of users. The customer is going to want a very simple Internet, a glitzy-looking type of application that will let him

get into specific functionality of these core business components. However, the clerk has to get a somewhat more sophisticated functionality such as car pick-up and credit approval. The manager needs to get the most sophisticated access of all—managing inventory and managing pricing. Everyone is using the same shared business components, operating under MTS control and going out to one or more databases that are distributed across the enterprise. So the same kind of architecture is very easily expandable in a way that typical rich-client applications are not.

Microsoft actually has some technology that's out in its early, immature stages called the repository. There is a tool that works on top of that called the component manager, and this is basically a tool to facilitate the broad use of these business components across different development projects. The real key to success in these kinds of distributed applications is taking a step back from this concept of a monolithic application and taking this huge three-year project and breaking it into a much smaller set of six-month deliverables. These large projects fail, and that gets proven over and over again; however there's a certain allure associated with putting everybody into a room and trying to get them to come out with the system. But as soon as I start to distribute both my application and the people creating that application, the need for communication and interaction between both the teams and the various components of the system is going to go up. So we do have a growing market of tools out there to facilitate the publication of business services that have been made available to the organization so that we're not duplicating work and using things in a consistent fashion. The tools are out there. COM is a facilitating technology because of the kind of self-documentation associated with its use. I can look at a component and get a very quick idea of how it is I avail myself of the services that it provides. So it's extremely important to do a little bit of extra management up front to manage the reintegration of these systems. We want to get away from this concept of stove pipes that we ran into with the database application. Therefore, we do need one integrated system at the back end. The answer is not to go all the way back to a monolithic system. Instead we need to be aggressive about the publication and integration of information between teams creating independent systems.

**From the Floor:** Human nature make us want to solve our own problem, create our own component, and pay to see a software solution. All that does is allow for proliferation of components as it does to consolidation.

**Mr. Zuck:** The question is about the natural proclivity to have each team doing their own thing. There's going to be a proliferation of components instead of the consolidation of components. The answer to that is simply going to come through proprietary access to information. Part of what we do when we build these componentized teams is give them rule over information. We move away from this

idea of shared data in which a lot of people are building similar components to form similar database operations; instead they don't have access to those data and they are forced to actually interact with that team directly to get at the information and the underlying processes that they need to move forward with that data. Much of what causes the proliferation is at the database level. You're suggestion of having kind of a component czar that keeps track of what components are being created is definitely a good one. In the practical environment of a large enterprise, it becomes very important. There's no question that there are going to be facilitators for centralized resources to facilitate the reintegration and the parallel development of multiple teams. It is important to maintain the autonomy of those teams. Part of what we do to facilitate that is control those data.

**Mr. Davlin:** Much of our work is very compute bound. We do have some heavy data files that we deal with, but generally we're bound on the numeric co-processor. I was wondering if MTS is designed to be well-suited to distributing numeric calculations as kind of a distributed compute server as well as things like data services.

**Mr. Zuck:** It is definitely a tool that will facilitate dynamic distribution of both compute services and data services. A key core functionality of MTS, that is sort of underimplemented at this point, is its capacity as an object request broker (ORB). They put in a lot of functionality and effort into the transaction monitoring; more so than they did into its ORB functionality. One of the key aspects of ORB functionality is going to be something called dynamic load balancing, which means that I can take an often-used calculation-type component and put it on multiple machines. Then I can have MTS dynamically decide which machine to go to based on which one is least utilized at the moment. Finally, I can substantiate a component on that machine. Right now, that is the more manual process, and it's not managed automatically by MTS. You can do something called static load balancing, which basically means that the customer and the inventory components are the ones most used. You're going to put those on two different machines and MTS will manage them all as part of a single transaction, but the ability to have the same component on multiple machines can dynamically decide where to go to and it's still a fairly manual process, but it's certainly one of the ones on the board for a coming release of MTS.

**Mr. Davlin:** I do have one comment on the size of the teams. A group called the Standish Group does different sorts of research in the data processing area. They claim that 53% of large software projects fail flat out. They analyzed the success rate by the size of the project in terms of how long it might take and how many developers are on it. They found that there was a sweet spot right at about four developers and four months. They thought that anything more than that would

cause you to run a very high risk of failure. They argued that if you're going to develop an n-tier system like this, you ought to try and partition the work load into different development teams of about that size and make assignments of about that scope.

**Mr. Zuck:** That is exactly what we did for a customer I was telling you about. He had gone three years without any productivity. The first thing we did was break it up into teams, and we had things going into data about six months later.

**Mr. Davlin:** Peter Donlon is going to discuss the experiences his company had moving to an n-tier architecture. He looks relatively unscathed, but I'm sure he has a few horror stories for us.

**Mr. J. Peter Donlon:** Jonathan has done a nice job of describing the technical foundation and the theory behind n-tier client/server architecture. What I'd like to do is talk to you from a business user's perspective. There are some challenges that we face within my company as we migrate from a two-tier structure ultimately to where we hope to be, which is an n-tier structure. I'd like to tell you about some of the benefits we've identified, some of the challenges we've already hit, and some of the tools that we've selected just to give you a sense of one example of a company going through the process.

It might help for me to just give you a little background about my company, which is slightly different than many of yours. You probably come from traditional insurance organizations. My company is USA Administration Services. We're basically a third-party administrator and system licensor. We're a subsidiary of Transamerica. Our business supported approximately 60 client insurance companies, but what's really critical with regard to our technology platform is that we need to have the ability to quickly put up new products. We have a wide variety of life insurance and annuity products that we support; in that sense, it's very similar to your own individual organizations. I'm sure you work with marketing areas that are constantly coming up with innovative new products with creative product features. You need to have the technology that can support the introduction of those products.

What might be helpful for you to know is this is our stated technology business philosophy, and I suspect it may not be all that dissimilar to what many of you are trying to accomplish, particularly within your information systems (IS) areas. A standard technical architecture is critical for effective business improvement and cost management. It's essential for consistency and continuity among what would otherwise be disparate technical development projects. Because we have so many technical development projects underway simultaneously, we needed a foundation

that could support all that complexity and all that variety. At the same time, we provide some continuity that would lead to an effective technical platform.

I thought it might be helpful to describe where we came from as an organization. In the early 1990s, we were focused primarily, like a lot of organizations, on the traditional two-tier model. That two-tier model basically has the presentation and the application logic all within a single component that communicates directly into the database layer.

By using that traditional two-tier model we found a couple of shortcomings inherent in that model that caused us problems in our business development. First, the traditional two-tier model was not very scalable. Second, it was difficult to ensure continuity. Third, there was very limited reuse. Limited reuse is one of the things that Jonathan talked a lot about with regard to one of the advantages of the n-tier structure. We found that a change in the presentation or logic area required a corresponding change in the other. That opened us up to creating a lot of errors. You make one change in a particular area, and it leads to other areas that you had not anticipated. We found that the number of users an application can support was a function of the database. All too often the number of users was limiting, and when you're a business like ours and you're dealing with multiple insurance companies and a large and growing number of users, you need to have a technology foundation that can support them. Finally, we found that as the application logic grew in size, a separate place was really needed to deploy the code. Merging the application code within the presentation layer was just workable for us.

We then moved to what we would label as a traditional three-tier model, and that model basically broke out so that we had separate layers for the presentation, the application logic, and for the database. In this model, the application developers had an additional challenge that we had not anticipated. We found that now they were not only responsible for the logic, but they were also responsible for the connectivity that had become even more important in this model than it had been initially in the two-tier model.

Then as we started thinking about how we could solve some of these challenges or problems that we were facing, we then targeted this concept of n-tier structure. We felt that this structure would give us some business benefits, and we could begin to partition the applications in a truly distributed environment. The three applications fed into a common application layer, which allowed a consistency of code, reuse of code, a great deal of scalability, and communication directly into multiple databases. We found that it's becoming increasingly important for our management reporting and analysis purposes to have access to multiple databases on various

systems operating in various operating environments. We needed a structure that could support that.

I thought it might be helpful to identify for you what some of the key drivers were for us as an organization to start looking at the n-tier model. We found that the n-tier model offered us a lot of attractions because we had certain business characteristics and we captured these in four major categories. You might find that some of these characteristics relate also to your operating environment. First, we believed n-tier was attractive for any organization going through combinations through mergers and acquisitions or accumulating additional blocks of business. As a third-party administrator, that's what we do every day, but I suspect that many of you in your own organizations are always looking at additional blocks to either sell or to buy in order to really focus on your core of strategic business strengths.

The second area was heterogeneous business environments with multiple products and services. We had to work in an environment where we were constantly being asked to support new and innovative products. Third was rapidly changing markets and products. Fourth was an increased need to obtain information from multiple databases. Many of our clients were demanding from us the ability to provide them with enhanced management information. That required us to communicate with not only our systems at USA but also systems at the client location.

The business benefits that we had targeted and that we expected to obtain through this migration to an n-tier model were reusability and scalability. This pertains to the reusability of the code, the application logic, and the scalability. As your business grew, we could simply add additional servers to accommodate that growth. There is ease of customization due to the modular component design of the n-tier structure. Another benefit is the ability to have a central, assessable customized application logic to reduce deployment costs so you're not actually deploying the logic to multiple locations. They can be geographically dispersed. It's all centralized. Simplified maintenance and reduced costs, the ability to ensure consistency in the development of your application, and the easy migration across systems and operating environments were all deemed to be critical for us.

I thought it might be useful to identify, from an actuarial perspective, how this type of technology might be useful to many of you in your work. There are certain advantages, such as looking at multiple databases for client information, that can be assessed using refined data-mining techniques. Many of you may be utilizing those techniques now. It can be used for experience studies based on mortality, persistency, and the ultimate impact that it will have on some of your new product pricing. The ability to identify key, unique characteristics for new customer classes is something that many of our clients claim is a primary concern. Finally, there is

the ability to identify and capture policy-level detail, which can be more easily consolidated at the company level for valuation or for facilitating reserving analysis.

As we started analyzing our movement to n-tier, we spent quite a bit of time evaluating the various tools that were on the market. Jonathan made mention of a number of the tools that were currently out there. I would not want to suggest that these are the right tools. I'll just offer this as an example of a company that has done some fairly significant analysis and tell you what we ultimately selected. At the tier-one level, we basically defined that as our thin presentation layer where we will have very minimal or no business logic located on the client workstation. It's this level that really communicates with the next tier, the tier-two component, and it's at this level that we utilize a graphical user interface (GUI). We use that for ease of use for our customer service representatives and for ease of training additional new employees.

One of the tools that we selected was Power Builder 5.0 from Power Soft. Many of you are familiar with that. There is also Microsoft Visual Basic, Windows 95, and NT. Connectivity is provided by dynamic linked libraries. Microsoft's RPC was identified as the tool to use to provide the necessary communication between the various levels.

The second tier is really where all of the application logic resides. It's located on a server or client workstation. It consists of reusable objects within libraries using distributed component, object-model standards. We needed to have some kind of standard that we could operate throughout our organization to ensure that we would have maximum efficiency in this tool. It provides for the essential connectivity between the other tiers—tier one, tier three, and ultimately, the n-tier. This level is responsible for the security and the performance and the scalability needs, and this was critical for us in handling an ever-increasing number of client requests. The application servers obviously must be extremely reliable. When the application server is down, so is the business, and the tools that we selected included Microsoft Visual C++, which we deployed across an NT server platform.

The third tier basically contains all of the relational databases, and these are accessible only through the second tier. This was a very useful method for us for segmenting our applications. It resides on a database server. Access is obtained through SQL and security is monitored and access is managed through a stored procedure. We selected a single Sybase 11 relational database, and it's written in Sybase stored procedure. Access to a remote, mainframe DB2 database is obtained through CICS .

The n-tier is what contains the common application logic layer shared by all the multiple users. It's important to have a highly efficient transaction service at this level. Component-based applications and component-based approaches to development are essential here. This was deemed really important in our efforts to move towards electronic commerce including Internet/Intranet, and other LAN applications.

The tools that we selected included Microsoft transaction servers, to manage the multiple access requests to a single database. So far it appears as though we've experienced about a 30% reduction in development time. We utilized DCOM, which is Microsoft's communication operating system, and that basically serves as a traffic cop for all the multiple requests coming in. Finally, remote connectivity is provide by TCP/IP networking.

As we made this migration from a two-tier up to an n-tier model, there are a number of challenges that we faced, which I thought might be informative. We saw pretty early on that not everyone can make this transition from a conventional object-oriented development to distributed computing. It's a very difficult change for a lot of people, and it required some skill sets that we did not have in our organization. We had to go outside for those. We found that some of the resulting bugs can be extremely difficult to isolate and correct.

We found strong object-oriented software engineering skills to be essential in our development process, but we found them very difficult to find. Our operation is located in Overland Park, a suburb of Kansas City. This suburb is a very strong, very large market for technology people, but we still had a hard time finding those skills. We ultimately determined that really few developers have a full understanding of building reusable components. Many developers too often view their work as art in progress. Everything is unique, everything is customizable, and the whole notion of reuse is not something that many of them find very appealing. We also found that it's important to recognize that not all applications should be moved out of a two-tier model. There are some applications that operate more effectively in a two-tier environment. We did find that it takes longer to create a three-tier versus a two-tier application. We estimated it to be anywhere between 30% and 40% longer. We concluded that it not only takes longer, but it's even much more complex and requires a much higher level of skills.

So those were the challenges, pitfalls, and problems we encountered. Our conclusion is that the benefits of reusability, the benefits of scalability (which were essential in our type of business), the benefits of high performance, and what we found openly to be relatively low cost compared to some other models, justified an investment in n-tier architecture.

**From the Floor:** Peter, I noticed you mentioned at the beginning that standards are very important. I agree because you could have chaos without certain standards. My question goes along with choosing development tools, especially when you're talking about a distributed environment, where you're using more than one tool. How do you come up with those standards? What kind of thought process do you go through to come up with those standards and decide on a suite of tools you're going to use from the services level right down to the data level?

**Mr. Donlon:** I can take the first shot at it. We have a relatively small IT shop within our organization. There are probably about 50 associates. A group of about five people within that larger group makes up our strategic system architecture group, and that's basically our think tank for people who spend a lot of time attending seminars like this or spend a lot of time in front of the various vendors of the numerous products. They spend a great deal of time negotiating with those vendors, and they have the ability to take a test drive on many of these products within our operating environment. It's a slow process. What we've learned is you really need to have a dedicated group. They tend to be our best performers from a technology perspective, and they're not just narrowly skilled technical people. They're very broad, strategic business people. Not only do you need a dedicated group, but I think you also need people who have a very broad and strategic perspective.

**Mr. Zuck:** I'd agree with everything Peter said, but it's also important not to go crazy in the production of standards, too. I mean, that can be another couple of binders of paper that people loosely adhere to. What's really important when developing standards is really standardizing the infrastructure and the methods through which different pieces of the applications are going to interact with each other. Choices about which development environment you use and things like that really aren't as important as they typically were in the past because if we standardize and modulate the ways in which components of the application interact, replacements of different application development environments are much easier than they've been in the past. Frankly, when you have a core set of tools that are capable of accommodating the infrastructure that you want to put in place, and if you end up making choices based on resource availability and education among the group that is trying to develop the application, you're always going to be faced with those kinds of realities. It isn't necessarily going to be practical to retool the entire organization for a particular tool or something like that. What's key is to make sure you use tools that are capable of participating in a well-structured architecture.

**Mr. Donlon:** I would actually like to add something to that as well. As I listened to Jonathan's comments, it reminded me of our environment or problem we had a number of years ago at our organization. I think we did err by trying to spend a lot of time and resources in identifying the right tools for various applications, and I think that was a mistake and a trap that we fell into. I think if you can convince your people who are doing the investigation that there is no single right answer that is appropriate for your organization and for your business needs, you'd probably be better off.

**Mr. Zuck:** The key is to keep in mind that you're trying to develop software for an organization that is a living entity. The best way to serve that organization is to deliver software solutions as quickly and as tightly designed as possible because both your IT solution and the organization you're trying to support have to be living organisms.

**Mr. Davlin:** You mentioned that when you went to the professional developers conference, there wasn't anything that really excited you because Microsoft is largely trying to convince people that there's something going on other than Java. Originally it started out as a failed language for programming toasters and things like that, and then it transformed into kind of the wooden stake to slay the werewolf. I never really bought into that aspect of it. The language is incredibly slow for things we'd like to do as actuaries, like numerical computations. Some companies like Semantic and Microsoft are providing native code environments for these to run more quickly; however, it's still not up to C speed. If you look at the language itself, its packages are really well suited for designing components for these n-tier kinds of architectures, and I was wondering if you picked up any information about Microsoft's plans in that regard. Is that the role that Java is going to fulfill in Microsoft's future?

**Mr. Zuck:** Part of what's going on is that Microsoft has in many respects taken over the presentational layer of the systems we're going to put in place. I mean, their market share in the desktop is so high right now that it is not even really a strategic market for them anymore. What it is really trying to gather up right now is that middle tier—application servers, application services, and so on. That's what really puts fear into the hearts of companies like Sun, where Unix had had traditional dominance. And so in many respects, Java could be compared to a game of Go. It is throwing a marble really far out on the board to cause some trouble while it tries to make more gradual encroachments further back.

There's a distinction between Java as a language and Java as a platform. The desire to use Java as a platform to somehow unseat Microsoft as an operating system vendor is ludicrous because the bottom line is this idea of dedicating yourself to

supporting interpreters on multiple platforms has come and gone many times in the past. The problem lies in the real, live, capitalistic market that we're in. This whole wonderful world of synergistic operating systems is never really going to exist. You have unity right now to break down the phone company. There was an attack on AT&T that took place; however, the result of that break up is people receive a lot of phone calls every day from every long distance company. I'm not convinced my rates are any cheaper.

At the same time, Java, as a language, is very elegant, and Microsoft has committed to facilitate the creation of these middle-tier components. I think that Microsoft views it as a good operating-system-specific language to facilitate the creation of core business components. It sort of combined transactional capabilities with more object-oriented designs. Much of what we saw at the primary domain controller (PDC) were improvements that Microsoft is making to these core infrastructure components. COM is now referred to as COM Plus, which is going to facilitate scalability through cached property assignments. One of the problems right now is if you take the same tool I used to communicate with Excel on one machine and suddenly move it across two machines without thinking about it, I could have an inefficient system with a lot of back and forth communications between those two machines. When we invented it for Excel, we weren't thinking about putting it on different machines.

The short-term answer is to redesign your components in a very transactional CICS-like fashion in order to minimize the amount of round trips across the network. Microsoft's next step is to improve that underlying COM architecture to do things like allow you to set the name, age, address, and phone number all at once. Then when you say, save, is it going to then set all those properties across and back again? Will it allow us the kind of flexibility we have of these rich object models, but still marry that to the requirements of a more scalable transactional system that we're going to need to create these three-tier solutions? I think Java is a big part of the answer. Java may, over time, supplant C++ because of its simplicity and its elegance. It's very different from this idea of Java as an operating system.

**Mr. Davlin:** Is increasing the native performance of Java a real priority?

**Mr. Zuck:** Definitely. Increasing the native performance of Java is a huge priority for Microsoft because they believe it is an operating-specific language. One of the key detractions from Java as an operating system is that it's really just another layer on top of an operating system. It is a tough sell to go into an organization and say, "You get to rewrite everything, and it's just going to run a little bit slower." People have trouble viewing that as progress, and that's constantly going to be the issue if the operating system gets improved independently of this separate, interpretive layer

in which Java is running. So a key part of Microsoft's strategy is to optimize Java for its operating systems.

**From the Floor:** Why did they go to this big system? They must have business restrictions sometime. Second, how long do you think this model has survived, and how much does the monitor cost? Will it be out of date in two years? What am I going to do about it?

**Mr. Zuck:** I'll respond to the second question about the core technology. This architecture is designed for that maintenance issue. You talk about building a system that will be out of date in two years. That's what this model is designed to address. Obviously I'm not a soothsayer. I can't say that three years from now this model will still be in force, but what I can do is look back across the history of application development and see where successes have occurred. Elements of those successes are part of what have brought us to where we are today. This isn't a completely revolutionary new thing that somebody just thought up like some fourteen-year-old in New Zealand who has supposedly cured the year 2000 problem. This isn't just some mystical solution. Instead it's an evolutionary outgrowth of a lot of years of experience.

I think that fundamentally, at a theoretical level, this model has a certain persistency to it and a certain evolutionary promise. When we develop an application for a business that's in a constant state of change, this is exactly the way we want to build it so that the system is a more accurate reflection of the organization as it's modeled at a business level. One of our clients is a government insurance firm, and the government is constantly changing the rules and the requirements all the time that are placed on this agency. We need to be able to make small changes to the system instead of having to redevelop the system as a whole. By creating a set of systems, each of those systems can be much more able to address changes to particular business practices. Rather than having some old legacy system on the line, it's kind of an all-or-nothing type of application.

I'm not sure which business obstacles you're referring to in general. Could you clarify your first question?

**From the Floor:** I'm referring to some kind of experience that won't evolve with those systems.

**Mr. Zuck:** We have that kind of experience all the time. An increase in complexity associated with an application of this type, which is part and parcel to developing your applications, is also changing the process by which you develop them. There are the kinds of teams that you try to form within the organization and the kinds of

time frames for deliverables that you try to set. When we talk about four-man, four-month deliverables, we're not outside the range of acceptability among the users. When we work for a customer, we try very hard for the very early and consistent involvement of the end user in the development process. We don't want to be part of a system in which the user develops some documents and then throws them over the wall to us so we can develop a system. The only way to get that level of user participation is to have this immediate return in order to shorten the amount of participation that they need to have, and to deliver a system to them in a short enough period of time so they feel that they're actually getting value for their time. There have been systems out there that have gone three years without anything being developed. That doesn't have to do with whether or not I did a two-tier or three-tier architecture; that has to do with the process whereby I integrate IT with the business organization, and it's that process that's really going to give us our time savings. It's that process that's going to allow this kind of a model to move forward. We've done it successfully and we have happy users as a result because they've been involved from day one to the last day. In the end, they have something that helps them do their job more efficiently.

**Mr. Donlon:** About five or six years ago in some of our early business planning work we defined one of our key organizational competencies to be technology. We didn't define what technology was. We just said it was going to be some form of technology that we recognized was going to evolve over time. Our migration from two-tier to n-tier is very much an evolutionary approach. We don't think it's a radical change, but we do think it's a change in which the value justifies the cost. I guess that would be a key challenge for any organization looking at a migration to any newer technologies. Make sure that some kind of rigorous business case is made because I do believe technology just for technology sake is probably not where any of us want to be.

**Mr. Davlin:** How much do these transaction servers and monitors cost? What does that add to development?

**Mr. Zuck:** Just the tools themselves?

**Mr. Davlin:** Yes, the tools themselves, the licensing issues, and things like that.

**Mr. Zuck:** The reality is that there are actually several tools of this type on the market. I'm trying to do a hard sell on one in particular, but some of them are designed as products that support someone's business model. In the case of the ((MTS), it's a part of Windows NT, so there's no cost associated with using MTS. It's also an evolutionary product that is very specific to COM and DCOM. If you really need the flexibility of being able to build non-COM based components on multiple

platforms like Unix or something like that, MTS probably isn't going to be the answer, at least in the short term. There are products out there like Encena or the newly released GTS from Sybase, which have a much higher cost associated with them. Using those products with all the knobs and buttons on the front can create a real nine-to-five job. MTS is really designed to be a more integral part of the underlying infrastructure of Windows NT. In a sense, it is designed to support a different business model. It's their support for a particular platform as opposed to a product by itself. But again, I consider all those costs to be insignificant compared to many of these peripheral costs associated with getting work done inside an organization.