Object Orientation Series: Principles of OO Technology

AGENDA

Object technology is here to stay. Understanding software development in the '90s requires at least a basic understanding of the principles and concepts of objects. Imparting that understanding is the goal of this course.

Hello, I'm David Chappell of Chappell and Associates. In this course - Principles of OO Technology - I want to start by giving a general introduction to the notion of objects. What are objects? What does it mean to write software using object technology?

Then I'll spend quite a while talking about the key object concepts and talking about the major ideas that underlie this approach to software development.

Then I'll talk about persistence, which means storing an object's data in some way. I'll talk generally about object-oriented development, say a few things about what this technology does, and about the development process. Then I'll talk about reuse, one of the key benefits that is often touted - and often achieved - by object technology.

In this program we'll also hear from Marie Lenzi , Managing Director of Syrinx Corporation and editor of Object Magazine.

APPROACHES

The Traditional Way

Think about how you build applications. Think about a traditional application. Typically, what you have is a sharp separation between code and data. You have the code, the actual instructions that you execute. As those instructions go along, as your program executes, data is read in, perhaps from some database management system or a file or something else.

As you march through your program, it reads in data appropriately and your program accepts and processes that data. There is, conventionally, a sharp distinction between the code and the data.

The OO Way

This distinction falls apart in the world of objects. With an application that is object-oriented, we think about the application. In fact, we build the application from a set of objects. Each object wraps together in one package, both code and data.

Our application is now viewed not in this wholly separate way of code and data. Instead our application becomes a set of cooperating objects. Each object talks to its fellows to get appropriate services. Each object encompasses both code and data.

Now if you're thinking, "Well, that data must still come from somewhere, from a database or a file or something else," you're right. We'll come back to that point. It's during execution that we visualize things as objects comprising code and data in one unit.

Look at a particular object, and here's what they actually look like. Objects have, as I've said, two major components. They have the data, also referred to as the object's state, and the actual code for the object. The object's code defines its behavior. It defines what the object can do. Typically, the object's code is broken up into some number of methods.

Methods

Now method is really a jargon word. Method really is a high-tech, object-oriented word for procedure or function or subroutine, in most cases. It doesn't always have to be, but usually that's what a method really turns out to be.

So each object can be thought of as a combination of state and behavior, as data and methods. Those two things wrapped together. The methods and object supports determine what that object can do.

The data the object contains determines what data those methods can act upon, typically.

Here's a simple example. Imagine an object representing a bank account, such as your bank account. This object has data and methods like all objects. Imagine this simple object has data like the name of the account owner and the balance of this account.

This object may have methods that allow a user of the object to credit a deposit or debit a withdrawal or get the name of the account owner or set the name, change it. They can perhaps query the balance and get the balance of this account object.

This example here illustrates the idea that objects contain data and they contain methods. Typically those methods act upon the data.

In another example, this bank with account objects very likely may also have objects for it's customers. So those objects will have different kinds of data and methods.

For example, the customer object might have data with the customer's name, the customer's ID number and maybe some indication of the number of accounts this customer has with the bank. The customer may, for example, have a savings account, checking account and so on.

The methods then for this kind of object might allow a user of the object to do things like add or delete a new account for this customer, get the customer name, ID number, and the number of customer accounts.

The point here illustrates this idea that objects are a combination of data and methods and that different kinds of objects will have different kinds of data and different kinds of methods.

Key OO Benefits

All this doesn't seem, perhaps, like a breakthrough. But we'll see there is certainly more to being object-oriented. But what we've seen so far allows us to talk about some of the key benefits of object technology. And that's important.

If this were just some new and different technology, that wouldn't be interesting. What's interesting is what this allows us to do.

A key benefit of object technology is that modeling problems as objects allows a much better match with the real world. In general this lets users be much more involved in creating solutions.

But you might say, "Well, wait, that's separation between code and data. That's a very natural model. We're all used to that model.'' And in fact we are. We've all written software for years. We know that model.

But our users don't. Our users think about things in terms, for example, of accounts and customers. They don't think in terms of separation of code from data. Modeling problems as objects lets users get much more involved in the analysis and design of those problems. This lets us create better software systems that better meet the needs of those users. That's important. That's a benefit.

Objects can also allow creating better code. By organizing your code in this way, you can produce code that is more maintainable and has fewer errors. We haven't yet seen the characteristics of object technology that allow these benefits to occur. We'll see them soon..

Finally, objects allow great potential for reuse, for creating a single object and reusing that code in many different ways.

Drawbacks of the OO Way

These are only some of the - I believe - key benefits of object technology. It would be unfair and unbalanced not to say a couple of things about the drawbacks as well.

The single biggest drawback of objects worth mentioning here is it could be complex. It can require retraining. Objects affect everything. Moving to objects is not just a matter of learning a new language or a new design technique It is all these and more. Moving to objects is a major undertaking. It is not a simple thing.

For most organizations, however, the benefits of doing so have greatly outweighed the drawbacks.

A Holistic Approach

MARIE LENZI: There are no applications that are better suited to object development or procedural development. Everything was done in procedural technology up until now because that's all we had. Everything looks like it fits in 80 column cards, whether it's a relational table or not, because that's all we have had. We have had to stuff the problem into our only solution space.

Now we have a broader, more flexible solution that offers us more dimensions. We have to bring our collective mind back to what the problem space looked like before we stuffed it into the 80-column-card mindset. Just about all applications are amenable to object technology.

There's a more holistic approach to the development process that the industry has embodied along with object technology. I'm not quite sure why or how that happened, but it is definitely a benefit to the development process.

People are not just looking at yet another compiler or operating system or tool. They're looking at the whole approach, at the meticulous definition of requirements, the architectural design of systems, the measurement and achievement of success - as opposed to just shoving in another piece of technology and letting it rip.

There's a lot that comes along with object technology and it's sort of coming along on the coattails.

The quick fix isn't going to happen. The expectations of the technology come when you look at it from a much broader perspective.

KEY OBJECT CONCEPTS

Classes & Instances

DAVID CHAPPELL: The first idea that's important to understand about object technology is the distinction between classes and instances. A class is really a template for a whole category of objects. An individual object is an instance of some class.

If I have two object instances of the same class, then those two objects will have the same methods and they'll have the same kinds of data. They will not, however, typically have the same values for that data.

The class defines the category. Each individual instance is an occurrence of that category.

Here's an example. Earlier I described the account object. What I really showed you earlier was the account class. Here are two examples of the account class, two instances of objects of the account class.

The first one represents the bank account of, let's say, Pamela Anderson Lee. So for the name field it says "Pamela" for the data. For the account balance it's got a hefty figure. She's doing okay these days.

It has the methods we saw earlier: credit, debit, getname, setname, getbalance. The same things that we saw when I showed you account earlier in the program.

The second instance of the account class is, let's say, for Mickey Dolenz. Mickey Dolenz, you recall, was the drummer in the Monkees, the world's finest rock and roll band. Mickey's not doing quite so well these days, so his account is somewhat smaller. Actually I hope Mickey's doing better than this because I think he's a very talented guy.

Anyway, this account object, this instance, has the same methods as the one for Pamela Anderson: credit, debit, getname, setname, getbalance.

What's different is the actual values of the data, two instances of the same class. This is an important distinction. It's an important thing to keep straight. The difference is between class - which defines whole categories of objects - and instances - which are actual examples of that class.

The Client

If I'm going to work with objects, and I don't use software, that means I am acting as the client of these objects. For a client to work with objects, typically the client must first create them. In most object-oriented languages, systems and so on, there is some mechanism provided which lets the client create objects.

For example, the client may have some new operator that the client can use to create, for instance, a new account or a new customer. The client, in this case, is creating instances of those classes. By client here, I mean software. The client may, in fact, be part of the same program or process as the objects, or it might be distinct, as in the cross-in network.

So client, in general throughout this course, means whatever code is using these objects. The client may, of course, itself be some other object.

The point here, however, is that clients typically must create actual object instances, always of some class, before they can work with them.

Messages

Once a client has an object to work with, the client can invoke the object's methods. Now, in the jargon of object technology, clients do this by sending what are called messages to objects.

Here, for example, we see a client invoking the debit method in some instance of the account class. The client does this by sending a message to the object.

Now message is really another jargon word. What actually happens here depends on the object system in use. Very possibly what happens is a function call. It's just a procedure call to invoke the method."
Alternatively, if the client and the object are across a network from each other, then it may be that this message is a remote procedure call. But generically the way a client invokes an object's method is referred to as "sending it a message.

OO CHARACTERISTICS

If I have classes, if I have instances, if I define my objects as having data and methods, am I then object-oriented? The answer, in the minds of most, is no.

For most people, there is really more required to qualify as object-oriented. There are three characteristics that most would argue are essential to be truly object-oriented.

Those characteristics are encapsulation, inheritance and polymorphism. I want to explain all three starting with encapsulation.

Encapsulation

Encapsulation is a blindingly simple idea. All it means is that the object's data is inaccessible to the object's clients except via the object's method. So for instance, if I have my client who wants to access, let's say the account balance of some account object instance, encapsulation says that the client can't read that data directly.

Instead the client must access, say, the debit method, by sending a message to the object. The debit method then affects the balance in some way.

Similarly, the client might want to access the name in this account, so the client will access the get-name method. This in turn pulls out the name from the object and sends it back to the client. If the client tries to access the name directly, it won't work. The data in the name is encapsulated. The client can't access it directly. This is a good thing.

This means that some kinds of errors that can occur by allowing direct access to data are prohibited. They are prevented from occurring by the principle of encapsulation. Now encapsulation really is just that: a principle. In practice, many object technologies allow ways around the straightjacket of encapsulation. But, in general, it's viewed as one of the three key characteristics of object technology.

Inheritance

The second characteristic is inheritance. Inheritance is a way to reuse something that already exists. For example, suppose I've defined some class, some object class. I wanted to find a new class. This new class may very likely have all the characteristics of the old class and a few more besides.

With inheritance I can do this. I can define the new class as a child of its parent. The new class will automatically inherit features from its parent, and perhaps add a few things as well.

Inheritance is a very nice way, in some cases, to reuse existing objects. Recall that word - reuse. A key benefit of object technology is reuse.

Inheritance is not the only way to reuse objects, but it is a very common way. It is one of the three aspects that most would consider essential to qualifying as truly object-oriented.

Here's an example. Think again about our account class. This class, you recall, defines objects that have name and balance for their data and have credit, debit, getname, setname, getbalance for their methods.

Suppose I want to define another class called checking account. Suppose that checking account ought to have all the data and all the methods of account, along with one more of each.

Let's imagine that this checking account class has the facility that, if a user would bounce a check, we will instead loan them money automatically. For this class, we can have an extra piece of data called loan amount, and one more method called get loan amount, which let's us query the object and learn how much is left to borrow. I don't know about you, but I'd be lost without this feature in my account. It's a useful thing.

What we're doing here is reusing at least the definition of the account class. In some cases, we may also get to reuse the actual code of the account class. Checking account can then easily, simply, be built upon the existing account class. That's inheritance. It's an easy and often very effective way to reuse existing classes and sometimes existing code.

Polymorphism

The third of the three characteristics that most would argue are essential to be truly object-oriented also has the most forbidding name. It's called polymorphism. What it means, however, is really quite simple.

Polymorphism means that a client can send the same message to two different objects, yet have each object run the appropriate method. This allows the client to treat objects of different classes as if they were the same. This makes life much easier for those clients.

The client does the same thing to each object, but each object behaves appropriately.

Here's a simple example. If I kiss my wife, she responds in a certain way. If I kiss a stranger on the street, he or she responds differently, I would assume. This is polymorphism. I am sending the same message to each object and each one is acting appropriately.

Here is a more professional example of the same thing. Imagine I have a client who is working with an instance of the account object and an instance of the checking account object.

If the client sends a message to the account object that says debit a certain amount from the account balance - clear a check, for example - then the account object will run its debit method.

The account object's debit method can be very simple. All it has to do is make sure that the current account balance is greater than or equal to the amount the client wants to withdraw. If it is, the account object processes the withdrawal. Simple.

Suppose, however, that the client sends the same message to the checking account object. It, too, might run it's debit method, but what it does is different. Recall that checking accounts have this extra feature of automatically loaning the client money, if required.

In this case, the debit method for checking account must check not just the balance, but it must check that the sum of balance and loan amount are greater than or equal to the amount the client wants to withdraw. If they are, it will process the withdrawal.

From the client's point of view, these two objects are the same. The client sends the same message, debit. Each object in fact behaves differently, appropriate to the object of its class. This is polymorphism.

That's it. You now know the key ideas and the key terms underlying object technology.

A QUICK OO REVIEW

A quick summary here.

Classes and instances. A class is a template for a whole category of objects and instance is a certain example, a particular instance of that class.

When clients want to talk to objects, when they want to invoke their methods, they do so by sending a message to the object. In practice this can mean a function call or something else.

Objects typically exhibit these three characteristics: encapsulation, which means the object's data is hidden from clients except via the methods; inheritance, which means objects can be derived from existing objects and hence reuse their code, or at least their definitions; and, polymorphism, which means that different classes of objects can accept the same message and respond appropriately.

This allows clients to have a much simpler view of the world.

There aren't many ideas here and yet these few ideas - coupled with the basic notion of an object, combination data and methods - have given rise to this whole technology, this very useful technology that we think of as objects.

PERSISTENCE

Earlier in this program, I made a point of distinguishing between conventional applications, which separate code and data, and object-oriented applications, which are built from objects - combinations of code and data, methods and data.

Well, the obvious question is where does that data come from? In a conventional application, you probably read the data from a database, for example, or a file. In object application that's just as true.

While the object is running, data is in memory, of course. But while the object is not running, that data must be stored somewhere.

In the jargon of objects, we say that those objects have persistence. They are persistent objects. Persistence just means that we store an object's state - it's data - between invocations of the object. Persistence is really a high-tech way of saying, "let's store stuff on disk."

For example, suppose I have an application built from several objects and those objects all have some data, like objects tend to do. At some point those objects must get their data loaded into them.

If the objects, for example, represent bank accounts or bank customers, then the data above those accounts and those customers is loaded from a database - for example - or a flat file. That data is from persistent storage, i.e. from disk.

Very commonly today that data is really stored in a typical relational database system: Oracle, SQL Server, DB-2. Pick your favorite.

Now there's an obvious problem here. The problem is that objects think about things to find abstractions in a certain way. Those abstractions don't exactly match the way in which conventional database systems store data.

A relational system, for example, stores data in tables. What do tables have to do with objects? Not a whole lot actually. You might store an object's data in a single row of a table and load that when the object is created. Or you might do other things.

The point is that the mapping between objects and a running program and data in a relational database is often imperfect. In some kinds of applications, the code that does the mapping can be a large part of the effort in building the application as a whole.

One way around this is to use a different kind of database. Rather than storing the object-persistent data in a flat file or a conventional relational database, why not store the object's data in an object database system? The idea here is to have a database system whose abstractions - whose approach to life, if you will - mirrors that of the objects in our applications.

Plenty of vendors sell these today. There is an active market in object-database-management systems. There is not, however, a terribly large market in this technology. It's out there, but, to date anyway, object-database systems have been used primarily in specialized applications.

Perhaps, as time goes on, as objects get more and more widely used, this will change. We'll see. For now what you want to keep in mind here is that object databases really are a way to do object persistence. They are not the only way, or even today the most common way. But they are certainly a way that fits well with the general paradigm of object technology.

THE FUTURE OF OBJECT DATABASES

MARIE LENZI: The future of object databases is not a technology problem. It is a marketing problem.

Interestingly enough you will find all of the database object leaders - that is the relational databases - have a definite and sore case of object envy. They are all desperately trying to impose object technology on their architectures. Some of them are going back and rearchitecting, an incredibly expensive and possibly prohibitive effort.

But recognize that all of the relational databases are, in fact, trying to get object-oriented in some fashion.

It looks like it will be a very difficult market for the object databases to address. I think their best bet, at this point in time, for obtaining some strong presence in the marketplace, is to go after distributed technology and web and intranet and Internet environments, where no-bit databases have an entrenched position. I think that may be their hope at this point in time.

Still it is not a technology problem. It is most definitely a marketing problem.

OBJECT-ORIENTED DEVELOPMENT

DAVID CHAPPELL: I've been talking for a while here about some of the narrowly technical aspects of objects: inheritance, polymorphism, persistence. I want to pull back a little bit and take a broader look now at what objects mean to the process of software development, and I want to start by talking about object-oriented programming languages.

Today, the most popular language in this space, by a wide margin, is C++. We also see a reasonable number of Smalltalk users, and more and more people are using Java. There are lots of others, but these three are the dominant players today.

It is very tempting, especially if you're a C programmer, to look at this and say, "Oh, well if I learn C++, which just extends C, then I've learned objects." It's easy to think that moving to objects is nothing more than learning a new language. This is not correct. Don't make this mistake.

As I hope you've seen - and you'll see throughout the rest of this program as well - objects entail a much bigger change in your thinking than just learning a new language. Should you choose to go to Smalltalk, for example, Smalltalk forces you to believe this because it drops you right into an unfamiliar world that is purely object-oriented.

While using OO languages can be a very good thing to do, don't be confused. Don't think that objects are just a new language. They are not.

To elaborate on this point, think about what happens when you design software. In the conventional world, you have code, you have data. You organize your data. You organize your code. All these things work together.

But if you're building an application from objects, now you have to think in terms of objects. What this means is you must sit down and decide what classes you want to have. You must decide how those classes relate to each other and how those classes interact.

For example, in our simple bank illustration, we might have an account class and that account class might be the parent class of two others called, let's say, checking account and savings account.

These two classes inherit from account and thus automatically get many of it's features. This same bank might also have a customer class and might - from that - define two other classes that inherit from customer called say, commercial customer and individual customer.

Deciding what classes you ought to have, who should inherit from whom, what those classes should do, what their methods should be and what is their data is not simple. Doing this requires the participation typically of end users. It is they who understand the business. It is they who know what kinds of customers and accounts or whatever you actually need to make your software match their problem.

As I said earlier, this reality that objects better match the world of our users is one of the big benefits of this technology. It allows those users to be more involved in the process of building the systems that we'll use for them.

To allow this, to support this whole idea - to think about our problems in terms of objects - there have been a number of methods created for object-oriented analysis and design. Among the most well known are, Ivar Jacobson's Objectory, the Object Modeling Technique from Rumbaugh, Grady Booch's Method and there are lots more. In fact, these three have been combining theirs into a unified method.

There are a number of ways to approach this problem of analysis and design. The key point here is to remember that moving to objects means that you need to change how you design, how you analyze, how you create your software solution.

The Waterfall Model

There's more. In conventional, traditional software development, people follow the waterfall model. They start, for example, with requirements analysis. Once this is done, it's passed on to software designers, who in turn pass the design to coders, who pass it to testers, who pass it to users and so on. Each step is commonly completed before the next one begins.

In this way, you can produce effective systems, as we know, but it tends to be somewhat static, somewhat inflexible. In a world of objects, it is possible to practice a much more iterative approach to development.

Iterative Development

With objects you can start by building the basic shell of what you're doing. You build a prototype that has the key objects and methods of those objects implemented. You can then experiment with that and build on that.

You can grow this, then, by adding more and more of your objects or building more and more of the methods. Over time you can create the complete solution.

You no longer have to have these sharp boundaries between the phases. You can instead have this more iterative process that grows more naturally out of your problem.

One of the drawbacks here, of course, is it can be hard to know when you're done, but still you could figure this out.

Objects change everything. They don't just change the language you use. They change analysis. They change design. They may very well change the way in which you create all of your applications.

Deciding on the Right Tool

MARIE LENZI: Sometimes picking the right tool for the right job can be a little difficult.

One side of you wants to pick the coolest, the niftiest, the slickest tools. The other side - the more practical side -will force you to look at the problem you are trying to solve.

But also, and equally as important, you look at the people who are going to be using these tools to solve the defined problem.

They are of equal weight and importance. If the problem defines and demands a certain set of tools, and you don't have the people who are able to wield those tools, then it is imperative that you either bring those people up to speed on those tools and get them some experience before they start working with those tools. Or get some people involved in the problem- solution space who are experienced with those particular tools.

When I say tools, I don't necessarily mean things that you're banging on the machine. It may be a design tool. It may be an architectural approach to systems development, as well as a compiler or development environment that you plug into the machines. It is not an easy problem to solve or differentiate.

REUSE TECHNIQUES

DAVID CHAPPELL: Earlier in this program, I said that one of the key benefits of object technology is the potential for reuse.

Now sometimes reuse can be oversold. Reuse is, in fact, quite hard to achieve. The barriers to reuse, while sometimes technical, are more often social and cultural and personal.

Still it is worth talking about some of the key technologies that are used to make reuse possible in the world of objects.

Class Libraries & Frameworks

A big idea is class libraries. A class library is just what the name suggests: it's a library of classes. If you, for example, have written some number of objects in C++ that you want to reuse at some later point, that might be useful for all sorts of applications and not just your own, you might package those objects into a class library. The idea here is that others can use your library and can perhaps inherit from the objects in it.

It's an effective way to reuse code in many situations. And today many people use - especially in C++ - class libraries written by others.

Over time, as class libraries evolve, their creators often find ways for the objects in those class libraries to interact, to use each other. The result is what is commonly known as a framework.

Now the framework is not very well defined. It means lots of different things. What I mean here - and I would submit as a very common meaning for this word - is a bunch of objects that all work together to provide some cohesive organized services. Frameworks are objects that generally are oriented towards some certain problem domain.

The idea here is to exploit the synergy that can emerge from a group of objects all built to work with one another. The idea is to make the whole greater than the sum of the parts.

Class libraries and frameworks are all well and good for object reuse. However, class libraries and frameworks both tend to have linkages to a certain language. Commonly for example, if I have objects in C++, I can reuse those objects only in other C++

programs. It would be great if I had some way to use objects in a different language once they're written in C++.

It would also be great if I had a way to distribute objects as binaries, not as source code. Class libraries are often sent out as source code.

Components

Both these problems, and more, are addressed by components. Now once again, the notion of component is very ill- defined. People use this word in a variety of ways. What I mean here are objects that allow cross-language reuse and binary distribution.

The notion is that with components I can create solutions, like assembling a puzzle. I can glue together discreet binary chunks of code, without regard for the language they're written in, to create my final solution.

There are, today, two leading architectures for building components. The first, and the most widely used, is Microsoft's Component Object Model, Microsoft's COM. COM underlies Microsoft's ActiveX and OLE technologies and is used extensively in Windows and Windows NT applications and also in those operating systems themselves.

The second technology in this area is IBM's SOM, the System Object Model. SOM underlies OpenDoc, a competitor, in some ways, to some parts of OLE. SOM also provides a way to create binary reusable components. Both of these technologies address the same set of problems and both have their adherents today.

The Distributed Environment

One more thing. If I'm going to reuse objects, or, for that matter, if I'm going to use objects at all, who says that those objects must exist on the same machine as the client? They don't have to. Why can't I access the services, the methods, of objects across the network somewhere?

There are today two leading technologies competing for dominance in this area as well. The first is from the Object Management Group, OMG. It's called CORBA, the Common Object Request Broker Architecture.

The second is an offshoot, an extension to Microsoft's COM, and it's called Distributed COM, or more commonly DCOM. CORBA and DCOM address similar problems. Both allow clients to invoke an object's methods across a network.

In fact, technically, the two have a great deal in common. However, unsurprisingly they are promoted by different organizations, who are, of course, competitors. Both of these technologies appear to have a role to play today. We'll see what happens in the future. We'll see if one wins out over the other. Today CORBA and DCOM are both powerful forces in this market.

Reuse happens in all sorts of ways. In a simple model, reuse happens just through inheritance. In a more complex environment, reuse happens from class libraries or frameworks or components or, perhaps, distribution. Reuse is important however it happens, and it is one of the benefits that people have achieved from object technology.

Component Technology & Reuse

MARIE LENZI: Component technology is a higher level of abstraction of reuse. So in order to achieve it, you're going to have to encompass all of the previously defined requirements for reuse - the broader perspective on development than just looking at the compilers and small technology - small-parts reuse.

Now going hand in hand with that are issues of scale. Component technology essentially is focused on a larger scale of reuse than just reusable parts. So keep in mind, when we go after that component technology that the scale is an incredible issue. True return on investment from component technology is going to be perceived at the business bottom line, not at the reuse of a button or a box or a window.

Reuse Is an Attitude

Reuse has been the biggest marketing mechanism and hype of object technology. However, reuse is not a technical problem. Primarily, reuse is an attitude. Reuse, too, has many dimensions. The least of which, in fact, is technology. Yes, we need technological elements to manage and control all the small components, but in order to achieve the small components, we need to look back at that holistic approach to development that I talked about earlier.

We need a model. We need a plan. We need a process and we need people who are willing to work, to achieve reusable components. You don't just cobble up a reusable component. It takes at least four or five hits on a component before it is, in fact, reusable across a larger dimension. It takes work and understanding, and it takes a cooperative effort of the development organization along with the business organization to achieve the reusable components.

Little buttons, windows, screens are all very well and good, and they will provide you some level of reuse. But not a real productive level. You need to look at larger componentry to get a real return on investment with reusable technology.

IN REVIEW

What are the two key parts of an object? Methods and data.

What's the difference between a class and an instance? A class is a template defining a general category of objects. An instance is a certain occurrence of a class.

What are the three typical characteristics of an object oriented technology? Inheritance, polymorphism, encapsulation.

What term is used to describe storing an object's data on a stable storage medium such as a disc drive? Persistence.

What is the most widely used object oriented programming language? C++.

THE EDUCATION PROCESS

MARIE LENZI: I have had one customer, when I walked into their office, who looked at me and said, "Marie, these people have not been trained in 30 years. Literally."

For whatever reason, our industry has a tendency to shun learning - painfully. And this is one industry where it is probably most imperative because it is constantly changing and advancing and some of this stuff we're making up as we go along.

Whether it be object technology, client/server, distributed computing, a new word processor, it is imperative that we focus more as an industry on this whole education process.

ADVICE

I think some of the best advice I can give is fairly simple advice - that is balancing reality with expectations.

The notion of "without a line of code" or "painless technology" is not a reality. We've been developing systems long enough to know that that kind of push-the-button magic is not going to happen.

The notion of going back and not only looking at the tools, but looking at the whole development environment and the development process, and, in fact, the relationship with the user community is also highly recommended. Look at a broader picture and improve your whole development effort, and you will recognize that this is not only applicable to object technology, but to any new technologies that you're going to be moving into your organization.

SUMMARY

DAVID CHAPPELL: So how important are objects? This is a heck of a time to ask. You've just watched the course, but the answer to the question is they are pretty darn important. It is possible today to build applications that don't use object technology. It's quite common, in fact.

It probably won't be in the future. It turns out that we are seeing object technology infiltrating the tools that we use. The tools we use to build applications, more and more, have at least some of the features of objects. The result is object technology is getting hard to avoid. The reasons for this are, as we've said throughout this program, objects provide a better way to model many problems.

Object technology can produce more reliable, more reusable software, and the result of this is that object technology is going mainstream.

A final cautionary note, this is not a panacea. This is not the longed for silver bullet that will solve all development problems. It is not. There's a steep learning curve here.

Nonetheless, the benefits of objects have outweighed the drawbacks for many, many people, and perhaps, they will for you as well.

I'm David Chappell. Thanks for watching.