Component Technology

Lars-Ola Österlund

ABB Automation Systems, Västerås, Sweden

In the software industry, the problem of creating modular systems has existed since the first systems were created. As systems grew larger, the need for modularization increased. Many different solutions have appeared, but none has solved the modularization problem well. The dream was and is to get software modules that can be installed like LEGO blocks or electronic integrated circuits. Object-oriented programming (OOP) was thought to enable software components, but it did not, at least not on its own. Enablers for component technology or component technology together with OOP turned out to be flexible interface definition languages, run-time binding/loading of executable code, and communication network infrastructures. A number of new such technologies have appeared that will take the software industry another step toward creating pluggable software components. This tutorial provides an overview of important technologies enabling component software and examines where component technology stands today.

The way a system is modularized into components is important, and many requirements must be met:

- Development and maintenance requires modularization of source code such that concurrent work is supported. It is no longer possible for one single organization to have knowledge enough to develop and maintain large software systems.
- The need to add new functions or upgrade already installed functions requires functional modularization. In large software systems, parts of the software often need to be updated due to corrections or functional enhancements. It is highly desirable that this does not affect the parts of the software that are not updated and that it does not interrupt the system.
- For stability reasons, software is divided into executable modules that execute isolated from each other. A crashing or misbehaving function is not allowed to disturb other functions or, in the worst case, stop the whole system.
- For scalability and performance reasons, software is divided into executable modules that can be deployed on one or more computers. When data volume and data flow increase, computers might be overloaded, which is a condition that requires modularization to distribute the workload over several computers.

The key to achieving successful modularization is to identify what functionality exists and to give this functionality well-defined interfaces. Once functions and interfaces are defined, they can be implemented by one or more components. Modularization also requires that the information handled by different components supports the modularization made. Each function will have its own information model. The part of function-specific information that is needed or used by other functions is exposed to them through an interface. An infrastructure for running components is required. At the most simplistic level, what is needed is support for exchange of data between running programs and a way for executing programs to activate other programs.

This tutorial starts with a discussion from this simplistic view by describing basic concepts, continues with a discussion of middleware of particular importance for component technology, and concludes with a discussion of what component technology will mean for applications.
Basic Concepts
Some basic terminology and concepts must be clarified in order to understand component technology.

Modularization and APIs
There are two basic views on software when creating a modular system. The first view is the functions and their interfaces. This can be called a functional view and describes what a system is expected to do and how to communicate with it. The second view, the file view (or implementation view), is how a program is divided into files. Most programming languages have two or three types of files. For compiled languages such as C and C++, they are source, object, and executable files.

In this tutorial, the word program represents a sequence of source code language statements. The long-established way to create boundaries between functions is the routine or subroutine (a routine is also a program). The word routine has slightly different meanings, depending on the programming language concerned. In this tutorial, it is also used to cover words such as function, method, coroutine, etc. In a routine, it is possible to hide any arbitrarily complex functionality, while the call to it still can be very simple. Its name and one or more formal parameters describe the routine interface. Examples in C are:

```c
int foo(void)
float foo(float inparameter1)
```

The routine name, together with the parameters, is often called the signature. In the example, the two foo routines are different. As they return different result data types and have different parameters, they have different signatures. The encapsulation of some functionality often requires more than one single routine but a set of routines. Such a set is often called an application program interface (API). With object-oriented technology, the set of routines is sorted into classes, and the API then becomes a set of classes. In classes, it is common to call the routines for methods or member functions. Such interface classes only describe the signatures and do not contain any implementation or code. So modularization of functionality is the problem of assigning functionality and APIs to routines.

In older systems, programs are compiled into a library of object code that is statically linked to executable modules. A change in a library routine requires the relinking of all executables using it, even if the API is unchanged. In a large system, this can cause a heavy maintenance burden. For this reason, it is desirable to use dynamic link libraries (DLLs) containing executable modules that are loaded at run time. This means that the executable modules in the library are loaded into the process when the API is called. As long as the API remains unchanged, it is possible to replace modules in the library without recompiling the calling program.

To run programs in a computer, the operating system loads its executable module into the primary memory and hands over the execution thread to it. The memory, together with the execution thread, is often called a process. A large software system normally consists of many executable modules generating many processes.

Figure 1 shows different types of software modules for compiled languages (e.g., C, C++, FORTRAN, etc.) and how they are related from source code to a running process. The boxes describe activities, and the storage symbols the file types. To perform the functionality expected from the system, processes cooperate by activating functions in each other and by the exchange of data. However, the modularization into processes comes with a price. As data cannot easily be shared between processes and execution threads cannot cross process borders, a communication mechanism is needed. Software facilitating such communication is often called middleware. The modularization into executable code is important, as it decides the granularity in which it is possible to add, update, or remove software.

Modularization in the file view is the problem of deciding what routines go into what file, how many executable modules go into a DLL or executable file, and how many processes are created at run time.

![Figure 1 Flow from program source code to executing process](image-url)
The announcement procedure moves the server awareness of clients from design time to run time. This is important, as it increases the decoupling between client and servers, thus giving better modularity.

Information Models

The processes exchanging data must have the same idea of what the data looks like. They must know the data types, what the data means, and what to do with it. They must use the same information model. An information model is usually described in terms of classes (or types), attributes, constraints, and relationships. Information models are expressed in specific languages, e.g., the unified modeling language (UML), object modeling technique (OMT), EXPRESS, SQL, etc. Recently, two new modeling languages have been presented by the World Wide Web Consortium (W3C): extensible markup language schema (XML) and resource definition framework schema (RDFS).

The types of data in an information model can be simple (e.g., integer, float, character, etc.) or complex. Complex types are called differently in different modeling languages. In EXPRESS, they are called entities, in UML/OMT classes, in SQL tables, and in RDFS resources. The meaning of a data type is usually to some extent captured in the name given. For example, if weight is measured in kilograms, a data type weightInKilograms can be created from an integer or float. As weight is a common property for many objects in the real world, the type weightInKilograms can be used as a type for an attribute weight in a number of different classes. When modularizing a system into components, an interesting question is: What happens with the information model? Is it modularized with the components? Will it be modularized independently or will it stay as one model? This will be discussed further.

Data Exchange

Processes generally do not share the same primary memory. An API is used to transfer the data into a client’s memory space. Data can be passed over the API either as typed parameters or in a buffer. For buffers, the data must be extracted from the buffer before use and stored into the buffer before transfer. Passing data through typed parameters can be made in two ways:

- By specific routines that can handle different information models. An example of such a routine is `getAttributeValue` (in `className` class, in `attributeName` attribute, out `valueType` value). In a client, a get call for the attribute `AttributeOne` will look something like: `... = getAttributeValue("MyClass", "AttributeOne")` assuming the `className` and `attributeName` are strings.

The content in a buffer is usually described by a separate description. Two basic approaches for this exist:

- An information model telling what data types it contains and how data is packed describes the contents of the buffer. The information model, together with data type sizes, is used to access the buffer. This way of describing and accessing data from buffers is used with relational database management systems (RDBMS), and the open database connect (ODBC) is such an API. Record-based formats, where the location and size of data within the record is separately described as in power system analytical data (PSAD) format, is another example.

- The buffer contains character data according to some grammar. To read the content in such a buffer, the content is first checked against the grammar such that the data is formally correct. This is called parsing. This way accessing data in buffers is used for XML data. The actual data is then described by a so-called document type declaration (DTD) that contains the rules used when parsing the data.

Compiling the information model into the API has the drawback that a change in the information model requires recompilation of all modules using the API. As information models likely will change over time, such an API will make a system hard to maintain.

Activation

The selection of a function to activate can be made either as a set of routines dedicated to each function or a general execute routine taking the selected function as a command parameter. Examples of commands used in SQL are select, update, insert, and delete. The decision to compile the select function into dedicated routines or a general execute routine is a tradeoff between flexibility and performance.

The initiative to actually transfer data can come either from the client directly calling data access functions in an API or from a server making a callback to a client. The first case is often called pull mode, and the second push mode. The pull mode of operation is efficient to use when a client needs data with a low frequency compared with the data update rate. In this case, most of the data has possibly changed, and it is for this
reason simpler to read all data each time it is needed. The push mode is efficient when a client needs data with a high frequency compared with the frequency data is updated with. Instead of reading all the data each time it is needed, the client can keep it in a local cache and ask the server to send or report changes only. This is often called publish/subscribe. To decouple the publisher and subscriber, the publisher makes an official publication of what data it has available and the subscribers establish subscriptions with the publisher of what data it wants. OLE for process control (OPC) is an example of a publish/subscribe interface. Client processes working in pull mode reading data may do so without necessarily being synchronized with the server process. If data is updated in the server process at the same time a client is reading, this may result in inconsistent data retrieved by the client. This calls for means to synchronize reads with writes such that data integrity is kept. To synchronize accesses, a mechanism called transactions can be used. Transactional access means that data changes made due to some event are collected into one single update operation that is not interruptible. This means that updates are made in sequence (they are serialized) and carried out without locking up the database for long periods of time. This, in turn, makes it possible for many clients to read the database concurrently. Publishers working in push mode will deliver new data to client processes by calling them directly. As the publishers and subscribers run asynchronously, this may mean that subscribers are not able to keep up with the speed in which the publishers are producing data. If the subscribers are not allowed to lose data, it must be temporarily stored in a queue and delivered to the subscribers when they are ready to take care of it.

Middleware

Middleware is software that is used to convey data between processes transparent to where data is located. This means that data might be located in any remote or local computer. All middleware has an API that is well defined and isolates the user from implementation details and the communication needed to get local/remote transparency. There are three kinds of technology that commonly are used as middleware: databases, message oriented middleware (MoM), and object request brokers (ORBs). Usually, these technologies are built on top of other middleware technologies, such as networking like TCP/IP, sockets, remote procedure calls (RPC), distributed file services, etc.

Databases

The main purpose of a database is to store complex data persistently, providing efficient access to it. Databases also provide support for implementation of information models through some data definition language (DDL). Many databases have built-in support for local/remote transparency. Due to this, databases have also been used as middleware, and data is exchanged between processes by reading and writing in the database. The API used for accessing a database may conform to a standard like SQL and use an API like ODBC or OLE/DB. Most databases are passive and are mainly used to implement the pull type of data exchange. To synchronize access, most databases support transactions. A few types of databases exist. Major types are relational database management systems (RDBMS) and object database management systems (ODBMS). For real-time systems and demands, dedicated real-time databases (RTDB) have been developed.

For a RDBMS client, there are three different methods to make database calls: embedded SQL, stored procedures, and direct use of a call level API. Embedded SQL is processed at compile time and is hardwired into the code for this reason. Compile time processing is also supplier dependent. For these reasons, embedded SQL does not support component technology. A client can call a stored procedure through an API. It refers to one or more precompiled SQL statements that are executed when the procedure is called. In this way, programming-language-specific APIs can be created for stored procedures. A call-level API is used to construct and execute SQL statements dynamically and to access data in buffers used by the SQL statements. Well-known call-level APIs are ODBC and OLE DB. ODBC is a C-language API, and OLE DB is a COM API. The software implementing ODBC is called an ODBC driver and OLE DB a COM component.

The local/remote transparency in a database is supplier specific. Most solutions rely on using a LAN connection located somewhere in the chain between the client process and the database server. For implementers of ODBC drivers and OLE DB components, each implementation is specific for the underlying database and is reliant on what services it provides by the database supplier.

Message-Oriented Middleware

The basic idea behind a MoM is to support local/remote transparent push mode operation. This means that a MoM contains persistently stored queues where messages sent are temporarily stored before they are delivered to clients at the pace with which the client can read messages. Another important feature is that clients and servers do not need to be aware of each other at run time. They only need to know the MoM APIs and the queue they are going to use for message exchange.

Since a MoM must be capable of transferring any kind of data, messages are stored in buffers. The publisher fills the buffer with a message and tells the MoM when to send. When the message buffer has been delivered to the MoM, the publisher immediately
continues, and the MoM takes care of the delivery to the subscribers. Some MoM supports guaranteed delivery, which means that the message would not become lost. If no subscribers currently have announced interest, the MoM will keep the messages until a subscriber subscribes for the queue holding the messages. When this happens, the subscriber will immediately start receiving messages. A subscriber who subscribes to a queue that is empty usually has three options for activation when a message arrives:

- Block in the MoM and wait for messages to come (push mode)
- Continue to execute and poll the queue for new messages at suitable times (pull mode)
- Continue to execute and be notified of arriving messages by a callback from the MoM (push mode)

A MoM API supports packing and unpacking of message buffers, sending and receiving of message buffers, and definition of queues.

Object Request Brokers
ORB technology is both a middleware and a component technology. As a middleware technology, it supports local/remote transparency to clients. As a component technology, it supports the definition of APIs and run time activation of executable modules. An ORB is transparent to programming languages used for implementation, which means that components developed in different programming languages can communicate over the ORB.

The two currently dominating ORB technologies are component object model (COM) and distributed COM (DCOM) from Microsoft and common object request broker architecture (CORBA) from the Object Management Group (OMG). Both are based on the same principles but use different syntax for defining APIs. CORBA is a set of specifications issued by OMG. The different suppliers who are members of OMG develop implementations supporting the specifications. One of the goals for CORBA was system interoperability between distributed systems, which means the communication protocol has been standardized. On TCP/IP, this protocol has the name Internet Inter-ORB Protocol (IIOP). COM was originally designed to support the modularization of Microsoft Windows applications running within one single PC, but has evolved into being an ORB as well. Limited versions of COM/DCOM are now also available on other platforms, but full COM/DCOM functionality is still limited to Windows applications on Intel processors.

ORB IDL
A basic functionality of an ORB is to provide the capability to define APIs. This is made with a specific interface definition language (IDL). CORBA and COM IDL look slightly different. The IDL is object oriented, which means routines, now called methods, are collected in classes. The IDL is flexible and gives a freedom to define various kinds of APIs supporting data access, activation of functions, callbacks, subscriptions.

A common way to illustrate client-server relations through an interface is the lollipop notation (Figure 2). A small circle represents the interface, which corresponds to a class without implementation. The server implementation class is represented by a box connected to the interface circle by a straight line and the client implementation using the interface is represented by another box with an arrow pointing to the interface circle to indicate use by the client.

![Figure 2 Lollipop interface notation](image)

Component Life Cycle
With the use of IDL, it is possible to develop the server and client functionality independent of each other as soon as an API is defined. This means that separate groups or organizations can develop the client and the server independently, which is a prerequisite for components to emerge. The life cycle from the definition of the IDL to actually running a component includes a number of steps, such as: interface design, software development, software installation, and software execution. These steps from design to actually running a component are briefly described as follows.

![Figure 3 Software development steps](image)
server and client implementation classes). At development time, the IDL is used when developing the client software using the interface and the server software implementing the interface. As a result, from the development, one or more executable files are generated. These files can exist, at this time, in distant locations, which normally is the case if different organizations are doing the development. The IDL is used to automatically generate code (proxy and stub) that is used to establish remote connections between the client and server processes. At install time, the server executables are installed into the server computer and the client executables on the client computer. These computers can be different or the same. When the server executables are installed, their locations and names are also entered in a database called repository. The ORB uses this database to locate the server executable at activation time. At run time, the first action is that a client is being activated. At some point in the execution, the client needs the services provided by the server. The client then contacts the ORB. If the server is not already running, the ORB finds the server executable through the repository and activates it. After this, the client is given back the wanted server interface and can begin to use it. The actual connection between the client and server is deferred to run time, and the only design/development-time dependency is through the interface. The calls are translated into data packets that are sent to the stub. This is called marshaling. From the marshaled data, the stub makes the actual calls to the implementation. Finally, results are transferred back (marshaled) from the stub to the proxy and the proxy returns the results to the client. The proxy/stub adds considerably to the overhead compared to the in-process case. In the case of remote servers, the marshaling involves sending the data packets over a LAN or some other communication media, which, of course, increases the overhead even more. The proxy and its marshaling are automatically generated from the IDL describing the API. The performance achieved with this standard marshalling is sometimes insufficient. Replacing the standard marshaling with other middleware solutions can create more efficient solutions.

Figure 5 shows the principle architecture for encapsulation of a number of different middleware solutions behind the same API. The upper part of the figure shows how the server functionality is divided into two different parts, the server-side processing and the client-side processing. This means that the function performed by a server is spliced into two parts, one located in the server process and one located in the client, similar to the in-process server case. Client-side processing is now called in-process handler or intelligent proxy. The benefit with such a split of functionality is the performance gains that can be made. The drawback is the increased work implementing communication between the two parts. Communication can be implemented with custom marshaling or a wide range of middleware, such as a MoM, remote procedure calls (RPC), sockets, TCP/IP, etc., as shown in the upper part of Figure 5. The lower part shows how a distributed file system or database can be used as middleware. The calculation/algorithmic part of the server functionality is now located entirely in the client. The distributed file system or database handles the persistency and local/remote transparency. This shows how the same API can be used to hide the underlying implementation.

In addition to the basic ORB functionality, a number of interfaces and services are defined using IDL. Some such interfaces are:

- Dynamic invocation of interfaces. If the client does not know in advance what interfaces a server supports, there is a service allowing the client to get a list of the interfaces supported by a server. The client can select and call interfaces from this list. Generic editors generating dialogs dynamically from the interface description typically use dynamic invocation of interfaces.
- Object naming for finding objects by name and life cycle services for creation/deletion of objects.
- Persistent storage of data related to objects.
• Event services supporting the publish/subscribe push model. An event service may use a MoM as its underlying middleware, conveying the messages between the server and client.

• Transaction services supporting synchronized updates for a set of data.

![Diagram of middleware solutions](image)

**Components**

The basic idea behind component software is to enable independent development of software modules that can plug together without requiring reccompilation. This will allow customers to buy components from different suppliers, install them, and get a working system that gives the joint functionality provided by all the different components. This is still an idealistic view of what can be achieved with component software, and a number of problems are yet to be solved.

A component can be defined as a set of executable modules having a well-defined API and a set of requirements on the environment in which it is to be installed. Requirements might be a particular operating system and middleware. Requirements for other functions or services to be installed may also exist. Such functions might be the previously mentioned object naming, event services, data persistency, transaction services, etc. The complete environment, consisting of the operating system, middleware, and additional support functions, is sometimes called the **component execution system** or **component execution platform**. Both the CORBA and the COM worlds are extending their functionality to be complete component execution systems. CORBA specification is extended with new services, and COM is extended with similar services into COM+ that will be available in Windows 2000.

**Objects and States**

A component, as such, does not contain any data or state, and it consists of one or more executable files. When a component is activated, one or more processes are created. The processes may have one or more instantly initiated objects, and the objects may have their own data or states. Data may be persistently stored in a file or database. This separation of components and component data may cause problems between the component version and the file format version or database information model. A simple example is a word processor component and documents saved to disk as files. The problem corresponds to file formats not compatible with all versions of the word processor component.

What are objects provided by an interface? One viewpoint is that there is a server object managing some data. The data can be accessed from the server object in a number of different ways: as many smaller objects or as a data buffer with (like in OLE/DB or ODBC) or without access routines. The other view is that all objects appear as server objects themselves and hence are managed directly by the ORB and not by a server object. The later possibility works well when the objects are few and data is simple. For data described by a complicated information model containing relations, the server object approach is better for many reasons (e.g., efficiency, object reference management, etc.). Questions that can be asked on objects and data are: Is the object and the data the same thing or different? Is there a one-to-one relation between objects and data? Is the object persistent or is the data persistent? Without proving it here, the best approach is to deal with objects and data separately. Objects are transient things appearing in the memory space of processes, and data are long-lived persistent things that may be accessed through an object. This corresponds to the three-tier architecture used in Microsoft COM+ and the component model referred to by CORBA components. The three-tier architecture is further discussed later.

When looking at information models describing power networks, the number of real-world objects easily becomes large. Counting all signal points, measurements, breakers, stations, etc. easily counts to millions or even more. Instantly initiating all of these real-world objects as objects in multiple server and client processes requires unaffordable amounts of memory. With the three-tier architecture, such multiplication of objects can be easily avoided.

**Technology Yet To Mature**

The problem of decoupling components from each other is solved to a fair degree. Yet to be solved, however, is the interface version, contractual interfaces, error management, and maintenance. When a component is enhanced, it is likely that the interface will reflect some of the enhancement. When updating an interface, the old interface has to be kept as well, or else the replacement of the component will adversely affect clients still using the old interface. This means that the interface versions have to be the same, and a component that is intended to be backwards compatible has to support older versions of the interface. Now imagine a system built from numerous components, all from different suppliers. If backward compatibility of interfaces is partial, this may end in a complicated puzzle of matching components with compatible interface versions.
The APIs that can be specified in CORBA or COM IDL mainly concern the signatures of available routines (methods). To further describe what a server can expect from a client and vice versa, a few more things have to be specified. Such specification is sometimes called a contract and contains:

- Specification of valid input parameter values, called preconditions. An example might be allowed attribute names that are used to select attributes for access in a server. In this case, an information model describes the preconditions.
- Specification of valid result values, called postconditions. This is what the client can expect as results from a server.
- Valid sequences of routine calls and reentrancy. If an API contains a number of routines, it is usually necessary to call them in specific sequences to get meaningful results. If multiple threads hit a component, reentrancy has to be controlled.
- The amount of time and space that can be expected to be consumed by the component. A server that consumes too much memory and takes too long time can make a whole system stop functioning.

In systems built from components, requirements on error diagnostics and inspection support are higher than in conventionally integrated software. If component source code and development environments are available, both can be used to trace and correct an error without involving component suppliers. As it is unlikely that component source code will be widely spread, specific error diagnostic/inspection support is required from components. In a system with components from many different suppliers, lack of diagnostics/inspection support will make it difficult to assign responsibility to trace and correct errors to suppliers.

Yet to be developed is how contracts should be practically formulated and verified as well as diagnostics/inspection support for tracing errors.

**Applications**

An application corresponds to a software package that provides a complete and meaningful functionality to the user of the application. Applications have commonly been implemented as one single software package with a monolithic internal structure. With the emergence of component technology, applications can now be divided into one or more components. As many applications have similar functions built in, these functions now can be encapsulated as components that are shared among several applications. Examples for desktop applications are spell checkers, grammar checkers, and filters.

In a large system having many applications, it is common that applications exchange data with each other. Component technology provides well-defined APIs that can be used for the application-to-application data exchange. Thus, component technology enables not only modularization of monolithic applications but application and systems integration as well.

**Resources and Dependencies**

To successfully divide an application into components, the components must be self-contained in the sense that all resources that an application needs are well defined and kept together with the application, i.e., resources are owned by the application. Some resources owned by an application will also be used by other applications. This creates a dependency between applications. Resources only used by the owning application can be changed without restrictions. Changes of resources used by other applications must be coordinated with the using applications. To get self-contained applications and to minimize dependency, the extent of shared resources must be minimized. This conserves modularity and prevents changes to ripple through a system and hit other applications in an uncontrolled way. If different applications come from different suppliers, such rippling changes will require tight cooperation between suppliers to produce coordinated updates of their components.

Resources that an application provides and that other applications use are supported by an API and an information model. The requirements an application has on other applications as well as the component execution platform can be expressed in a dependency graph where supported interfaces match with required interfaces (Figure 6).

![Figure 6 Application dependency graph](image)

The arrows in Figure 6 mean a client application pointing at server application depends on the pointed-at application. Applications 1A and 1B both implement the same function. However, they have a small difference in the interface they support. Applications 2 and 3 implement different functions, but both require interfaces supplied by type 1 applications. As application 1B does not support the requirements from application 2, there is no dependency between these two. Application 4 is included only to indicate that an application (Application 2) can appear both as a server and client.

**Architecture**

Application architecture can be described for an application itself and relative other applications. A
simple model of how applications, components, and the component execution platform relate is shown in Figure 7.

An application is built from components, and components implement interfaces that are exposed to other components through the component execution platform. Which component implements what interfaces is an implementation issue. The wrapper connects legacy application code with the component interface implementation.

Internally, an application is usually built from three major pieces (Figure 8):

- An HMI part, which presents data from the application to the user.
- Some business logic, which performs calculations and computations.
- A data part, which is responsible for data access and persistency.

This division corresponds to the idea that objects and data are kept separately. In the past, the three pieces were intermingled and difficult to distinguish from each other. With the occurrence of the client-server technology, the HMI and the data parts were separated. Logic sometimes went with the HMI and sometimes with the data part. This corresponds to the so-called two-tier architecture. Now the logic is regarded as a separate tier resulting in the three-tier architecture.

In the three-tier architecture, clients are only allowed to make data access through the logic tier. This gives considerable freedom to how an information model exposed by the logic is actually implemented in the data store that is now private to the logic tier. This means that the data access interface exposed by the data store needs not be the same as exposed by the logic.

The server logic in one application may also use the API of another application. In the previous discussion on application resources, such a use corresponds to a dependency between the applications.

**Future Trends**

For SCADA/EMS systems, performance and scalability requirements are high. This has resulted in today’s optimized designs that give both high real-time performance and good scalability. The challenge for SCADA/EMS suppliers is to keep today’s good performance and scalability characteristics while making modular and componentized products. The key to getting good internal modularization of current systems as well as successful integration of systems is to define functions and interfaces.

Component technology is a strong trend in the software industry; both small and large systems will become componentized over time. This will facilitate more efficient use of development resources and create new component markets.

Problems like lack of contractual interfaces, control of resource consumption, and lack of error diagnostics/inspection support have to be solved before component markets will prosper. Meanwhile, component and interface technology will be used to integrate application packages before the application packages themselves are internally divided into components.

**Biography**

**Lars-Ola G. Österlund** is a senior analyst of software architecture and information modeling at ABB Automation Systems, Power Utilities Division, in Västerås, Sweden. He received an MSEE from Lunds Institute of Technology, Sweden, and a diploma on artificial intelligence and expert systems from Linköping Institute of Technology in Linköping, Sweden. Since 1975, he has been employed by ASEA and later ABB. ABB was formed in the ASEA/BBC merger 1988. Within ASEA/ABB he has been working with development of software in the fields of communication, EMS network applications, nuclear power plants, and applied artificial intelligence and expert systems. He has also been working with software production and maintenance process, including systems integration and testing. He has also managed R&D groups in these fields.
For Further Reading

"The component object model specification", version 0.9, Microsoft, http://msdn.microsoft.com/library/default.htm

"RDF/RDFS, W3C", http://www.w3.org/Metadata/

Abbreviations & Acronyms

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>application program interface</td>
</tr>
<tr>
<td>COM</td>
<td>component object model</td>
</tr>
<tr>
<td>CORBA</td>
<td>common object request broker architecture</td>
</tr>
<tr>
<td>DLL</td>
<td>dynamic link library</td>
</tr>
<tr>
<td>DTD</td>
<td>document type declaration</td>
</tr>
<tr>
<td>IDL</td>
<td>interface definition language</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet inter-ORB protocol</td>
</tr>
<tr>
<td>MoM</td>
<td>message oriented middleware</td>
</tr>
<tr>
<td>ODBC</td>
<td>open database connect</td>
</tr>
<tr>
<td>OLE</td>
<td>object linking and embedding</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OMT</td>
<td>object modeling technique</td>
</tr>
<tr>
<td>OOP</td>
<td>object oriented programming</td>
</tr>
<tr>
<td>ORB</td>
<td>object request broker</td>
</tr>
<tr>
<td>PSAD</td>
<td>power system analytical data</td>
</tr>
<tr>
<td>RDBMS</td>
<td>relational database management systems</td>
</tr>
<tr>
<td>RDF</td>
<td>resource definition framework</td>
</tr>
<tr>
<td>RDFS</td>
<td>resource definition framework schema</td>
</tr>
<tr>
<td>RPC</td>
<td>remote procedure call</td>
</tr>
<tr>
<td>RTDB</td>
<td>real-time database</td>
</tr>
<tr>
<td>SQL</td>
<td>structured query language</td>
</tr>
<tr>
<td>UML</td>
<td>unified modelling language</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>Wintel</td>
<td>Microsoft Windows OS on Intel CPUs</td>
</tr>
<tr>
<td>XML</td>
<td>extensible markup language</td>
</tr>
</tbody>
</table>