An Approach to Architecting Enterprise Solutions

A frequently mentioned ailment in healthcare information management is the lack of compatibility among information systems. To address this problem, HP's Medical Products Group has created a high-level model that defines the major architectural elements required for a complete healthcare enterprise information system.

by Robert A. Seliger

HP's Medical Products Group (MPG) produces medical devices such as patient monitors and ultrasound imaging systems, which obtain physiological data from patients, and clinical information systems, which document, retrieve, and analyze patient data.

In December 1994, MPG directed its architects to define and drive the implementation of an open, standards-based MPG application system architecture that would enable:

- Improved application development productivity
- Faster times to market
- Seamless integration of applications developed by MPG and its partners
- Integration with contemporary and legacy systems in an open standards-based environment

To meet these objectives and to help establish MPG as a leader in healthcare information systems, the Concert architecture was conceived. Concert is a software platform for component-based, enterprise-capable healthcare information systems.

The primary objective of Concert is to enable the decomposition of healthcare applications and systems of applications into sets of interconnectable collaborative components. Each component implements important aspects of a complete healthcare application or system of applications. The components work together to realize fully functional applications and systems of applications.

A component-based approach was pursued to leverage the fundamental precepts of good software engineering: *decomposition*, *abstraction*, and *modularity*. We reasoned that an architecture that facilitated decomposing large complex systems into modular components and abstracted the details of their implementation would contribute to development productivity. The ability to use these components in a variety of applications would expedite time to market.

Carefully specified component interfaces would enable flexible integration of components in a seamless manner. Openly publishing these interfaces would enable components developed by MPG's partners to interoperate with MPG's components. The judicious use of healthcare and computing standards would enable integration with systems based upon other architectures.

Concert was developed by MPG in conjunction with HP Laboratories and the Mayo Clinic, a strategic MPG partner. It serves as the technical cornerstone for MPG's group-wide initiative to provide better enterprise solutions for its customers. Key aspects of the architecture have also been applied by HP Laboratories and the Mayo Clinic to develop a prototype electronic medical record system.

Concert also serves as the foundation for the technical development effort of the Andover Working Group for Open Healthcare Interoperability. This MPG-led healthcare industry initiative was been formed to achieve enterprise-wide multivendor interoperability (see **Subarticle 2**).

Concert currently consists of the following elements:

- A general reference model that organizes the architecture of healthcare enterprise information systems into a key set of architectural ingredients
- A model for software components that can be implemented using CORBA-based[1] or Microsoft® OLE-based[2] technologies
- An initial set of Concert components including their interfaces and the policies that govern the patterns of interaction between the components
- An approach for organizing Concert component interfaces to represent component application development, system integration, and system management capabilities

- An initial information model that provides an object-oriented description of healthcare terms, concepts, entities, and relationships to establish a common clinical basis for Concert components and the applications developed from them.

Concert Components

To facilitate the description of the Concert component model, an example of one of the components that MPG has developed will be used. The component, called an *enterprise communicator*, is at the heart of the enterprise communication framework (ECF) that MPG is developing in conjunction with other healthcare vendors and providers that form the Andover Working Group.

An enterprise communicator is a software component that facilitates healthcare standards-based data interchange between healthcare systems and applications within a healthcare enterprise. Different types of communicators encapsulate different healthcare standards. The particular communicator that MPG is currently developing encapsulates the Health Level 7 (HL7) 2.2 data interchange standard.[3] Fig. 1 shows a system based on enterprise communicators.

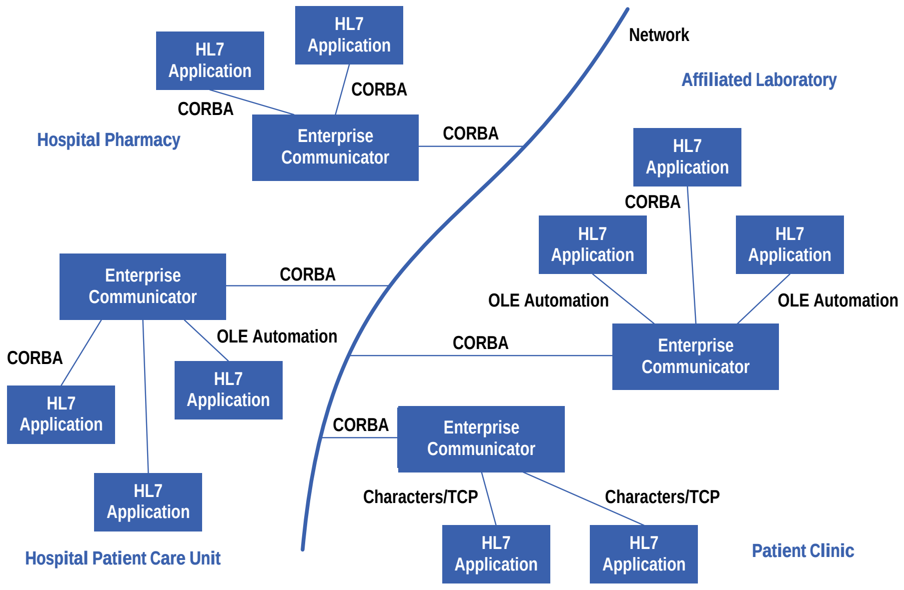


Fig. 1. *A healthcare system based on enterprise communicators and the HL7 data interchange standard.*

HL7 is a widely used healthcare electronic data interchange standard. Its primary contribution is the specification of a set of messages that healthcare systems can exchange to share clinically relevant data. Examples include messages that enable applications to obtain the results of laboratory tests from the applications that have access to this data.

The HL7 standard is not intended to be particularly prescriptive in terms of messaging technology or how messaging services should be implemented. This has led to a variety of custom HL7 implementations based on a range of technologies. A typical implementation employs specially formatted character-encoded messages and point-to-point network or serial-line connections. An example of a character-encoded HL7 message is shown in Fig. 2.

In the Concert-based model, applications employ enterprise communicators to broker their HL7 data interchange needs. Enterprise communicators provide applications with the necessary messaging capabilities, such as guaranteed message delivery and multicasting (i.e., sending several messages at once). Enterprise communicators also present HL7 messages as object-oriented abstractions using both CORBA and OLE automation technologies. This eliminates the need for applications to parse the messages to extract the encoded data.

In addition, for legacy integration, communicators support TCP/IP interfaces through which applications that are not object-oriented can send and receive character-encoded HL7 messages.

Why Components?

The concept of component-based systems has become increasingly popular over the last several years. There are currently many definitions of components and a variety of tools and technologies have emerged to facilitate developing component-based systems. Many of the general concepts about what a component is are similar across all of these definitions. However, there appears to be little agreement on the granularity of a component. Granularity depends on how much functionality a component represents and how much code and complexity are embodied within a component implementation.

```
MSH|^~\&|HP|HOSPITAL|AWG|GENERAL CLINIC|199608271353|
ADT^AO1|56844_1_AA|T|2.2|<cr>
EVN|AO1|199608271353|<cr>
PID|l| |102983|3106|DOE^JOHN^^SR^^ | |19450112|M|
AWG^TEST^^^^~ANDOVER^GROUP^^^^ |H|1400 MAIN ST^ ROOM 6263^ANDOVER^
MA^10810| |(508)555-1022|862-1022| |M|CAT|14563|838-29-4938|
<13>NK1|1|DOE JOAN B |CHD|101 MAIN STREET APT^2A^BOSTON^MA^O5404|
(508)555-0000|(508)555-0000|<cr>
NK1|2|DOE JANE^^^^|CHD|434 NORTH STREET^APARTMENT 5B^CHELMSFORD^MA^
05401|(508)555-1111| |<Cr>
PV1|1|I|1N^107 A|ELE| | |36^HEART^THOMAS^MD^^^| | |IMX| | | |1| |Y|
36^HEART^THOMAS^MD^^^|0|14563| | | | | | | | | | | | | | | | | | | | |
| | 1W^102^W| |199608191015|<cr>
PV2| |s| |<cr>
```

basically means ...

Admit patient John Doe Sr., whose wife is Joan and next-of-kin is Jane, and whose physician is Dr. Heart, to General Clinic

Fig. 2. *A character-encoded HL7 admit patient message.*

In Concert, components tend to be medium-to-large-grained objects.[4] For example, a Concert component might be implemented by what is traditionally thought of as an executable program, as is the case for an enterprise communicator. Alternatively, a group of Concert components might be packaged within a library. However, a Concert component is rarely as small as a single C++ or Smalltalk object.

In general, a Concert component is a portion of an application system that:

- Implements a substantial portion of the overall application system's capabilities
- Represents its capabilities via one or more modularly defined binary interfaces
- Can be developed independently of other components
- Is capable of efficiently communicating with other components over a network
- Is the fundamental unit of configurability, extensibility, replaceability, and distribution
- Is the basis for an open system through the publication of its interfaces.

In other words, a Concert component represents a significant portion of an overall application system, but is small enough to enable efficient and flexible composition with other components to form full-fledged applications and application systems.

A key motivation for a component-based architecture is that it makes accomplishing the following architectural objectives much easier.

- Simplification. Components can make the approach to decomposing a complex application system into smaller simpler pieces tangible and precise.
- Replaceability. Existing components can be readily replaced with new implementations as long as the new component supports the same interfaces as the component it replaces.
- Configurability. Components provide a modular, precise, and manageable basis for configuring a system.
- Extensibility. New components with new capabilities can be added to an existing system in a modular and organized manner. The risk of breaking existing capabilities that are well-encapsulated in existing components is minimized. In addition, new capabilities that are added to existing components can be represented by new component interfaces that represent the new capabilities without requiring changes to existing interfaces.
- Independence. The interfaces between components define the "contract" between components that can enable independent development as long as the contracts are respected.
- Scalability. Components can be physically distributed or alternatively collocated depending upon the computing infrastructure available and desired price/performance profile. The component interfaces define what components communicate about, and this communication can be realized using same-machine or network-based mechanisms.
- Stability. A variety of tools, technologies, and design methods can be employed to implement the components, thereby enabling evolution of the implementation technology, tools, and methods without violation of the architecture.
- Business-Centeredness. The efficient and timely realization of the architectural objectives listed above is the basis for a significant competitive advantage.

To achieve these objectives, Concert specifications primarily emphasize how application systems are assembled from components. This approach provides a great deal of latitude for application developers to define what capabilities their application systems will actually provide. Perhaps most important, the architecture also enables product teams to put more focus on developing the content of their applications because they can leverage a standard approach to constructing their application systems.

Component Interfaces

Concert components implement object-oriented interfaces. An object-oriented interface is a named grouping of semantically related operations that can be performed on a component. A component that implements a particular interface implicitly supports the functionality contract implied by the interface.

For example, among the interfaces that an enterprise communicator implements is the ApplicationConnect interface. This interface enables an application to connect to and disconnect from a communicator. Only connected applications can send and receive HL7 messages.

Components that implement similar capabilities represent these capabilities via the same interface. For example, any Concert component that requires its client applications to explicitly connect and disconnect might implement the ApplicationConnect interface. The effect of connecting and disconnecting would depend upon the type of component, but the policies governing when and how to use the interface would be the same.

Each object-oriented interface enables a subset of a component's overall capabilities to be accessed and applied. A component's full set of object-oriented interfaces enables a component's full array of capabilities to be accessed and applied. For example, another interface that is implemented by an enterprise communicator is MessageManager. This interface enables a connected application either to create a new message that can be populated with data and sent or to obtain a message that the communicator has received from another application.

Many of the details of the Concert model for software components come from the OMG's Object Management Architecture (OMA). The most notable OMA ingredient is the use of the OMG Interface Definition Language (OMG IDL) for specifying a Concert component's object-oriented interfaces independent of the technology used to implement the component and its interfaces.

OMG IDL serves as the software equivalent of the schematic symbols that electrical engineers use to diagram circuits. For example, the symbol for an AND gate clearly conveys its role without relying on descriptions of the underlying circuitry or fabrication technology (e.g., CMOS, TTL, etc.).

OMG IDL provides a standard and formal way to describe software component interfaces. Further, when applied within the context of an overall component-based architecture, formally specified interfaces can be used to create a level of precision that helps ensure that important architectural features and principles are reflected in products that are eventually developed. For example, components that constitute a particular product can be examined to see if they correctly implement the necessary interfaces. The role that interfaces play in adding precision to a software architecture is illustrated in Fig. 3.

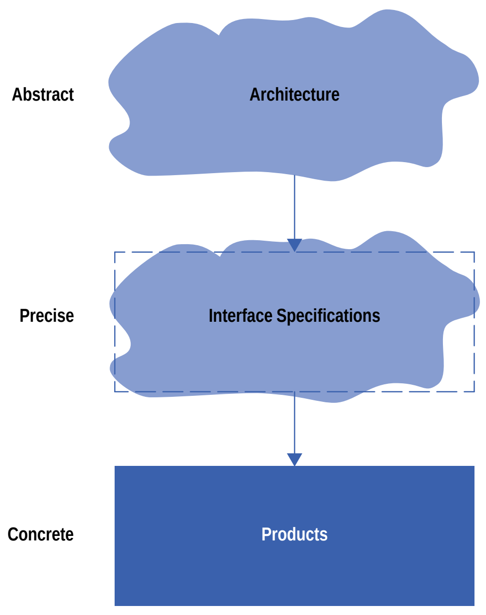


Fig. 3. *An illustration of the role that interfaces play in adding precision to a software architecture.*

Another advantage of defining interfaces is that they can provide a shorthand for describing components. The Concert specification currently consists of less than forty interfaces. Just the name of the interfaces that a component implements is often all one needs to understand how to use the component.

For example, the enterprise communicator interface ImplementationInformation allows access to implementation information about a communicator, including its product number, software revision, and when it was installed on its current host. The interface HostInformation provides access to information about the computer that is hosting an enterprise communicator, including the host's network name and the type of operating system it supports.

A simplified OMG IDL specification for an enterprise communicator's ApplicationConnect interface is shown in Fig. 4. This specification conveys the following information about the interface:

- The name of the interface is ApplicationConnect.
- The interface supports two operations: connect and disconnect. An application that wants to connect to an enterprise communicator performs the connect operation on the communicator's ApplicationConnect interface. An application that wants to disconnect performs the disconnect operation. In either case, the application identifies itself by setting an appropriate value for the input parameter which_application.
- Under normal conditions, neither operation returns any data. However, they can raise exceptions. An operation that has encountered an abnormal condition can communicate this fact to its client by raising an exception. When an operation completes, its client is able to determine whether or not the operation completed normally or has raised an exception.

```
interface ApplicationConnect : Composable {
 exception UnknownApplication {};
 exception AlreadyConnected {};
 exception NotConnected {};
 void connect (in ApplicationIdentifier which_application)
   raises (UnknownApplication, AlreadyConnected);
 void disconnect (in ApplicationIdentifier which_application)
   raises (UnknownApplication, NotConnected);
} ;
```

Fig. 4. *An example of an interface definition.*

Different types of exceptions can be defined, each of which represents a different abnormal condition. The exception UnknownApplication indicates that the application identified by the parameter which_application is not known to the enterprise communicator. The exception AlreadyConnected indicates that the application that is trying to connect is already connected to the enterprise communicator. The exception NotConnected indicates that an application that is trying to disconnect is not currently connected to the enterprise communicator.

Another important characteristic of the ApplicationConnect interface is that it inherits the definition specified for the interface Composable, which is described in the next section.

Multiple Interfaces. Additional ingredients of the Concert model for components were leveraged from Microsoft's Component Object Model (COM). While much of COM describes the low-level conventions for performing operation invocations on objects, COM also motivates the concept of representing a component through multiple distinct interfaces.

In COM, a client must explicitly ask a component whether it supports a particular interface before it can access the interface. If the component does indeed support the interface, the client can use it. Otherwise, the client must seek another interface, or try to make do with the interfaces that are supported. See Subarticle 3, "***Multiple Interfaces in COM***."

Although typically described by Microsoft as a way to evolve component functionality through the addition of new interfaces and as a way to simplify perceived problems with object-oriented inheritance, the real strength of multiple interfaces is the ability to model complexity.

For example, in Concert, components represent significant subsets of the overall functionality of an application or system of applications. It would be unwieldy to try to represent a component's complete set of capabilities through a single interface. It would be unnatural in many cases to impart modularity by organizing these interfaces using inheritance.

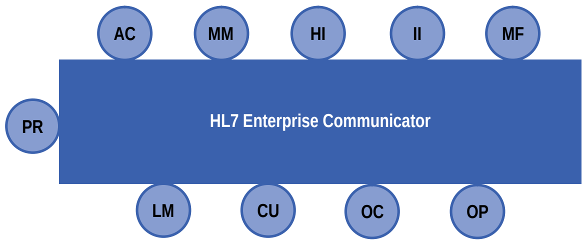
As a simple real-world example of multiple interfaces, consider the interfaces that might represent an employee who is also a father and a baseball fan. It is unnatural to model this employee's interfaces using an inheritance relationship because the interfaces are semantically unrelated. It would be awkward to define a single employee-specific interface because the advantages of developing distinct models for the concepts of the employee as father and baseball fan become obscured. However, modeling the employee as supporting multiple interfaces is essentially how things work in the real world.

Concert's adaptation of the COM concept of multiple interfaces is referred to as *interface composition*. This is because a component's functionality is represented by a composition of distinct interfaces. The interfaces that the component chooses to include in this composition can vary over time as a function of the component's internal state or because its underlying implementation has changed.

The interfaces in a Concert interface composition are referred to as *composable* interfaces. In Concert, all composable interfaces are derived from the base interface Composable. The interface Composable provides functionality similar to COM's IUnknown:. It supports a method similar to QueryInterface which enables a component's client to determine whether the component implements a particular interface, and if so, to obtain a reference to the interface. For convenience, this querying capability is available via any Composable interface.

In addition, every component implements the interface Principal, which is also derived from Composable. In addition to providing a way for a component's clients to interrogate a component about the interfaces it supports, Principal also enables clients to obtain a list of all of the interfaces that the component currently implements. For some clients the ability to obtain a list of available interfaces is preferred to the technique of interrogating for interfaces one at a time.

The primary difference between COM and Concert in terms of support for multiple interfaces is that in COM, the concept only applies to components implemented using COM-based technology. In Concert, the concept has been easily layered on top of a variety of technologies, including COM, CORBA, and even Smalltalk and C++. This enables Concert to apply a powerful architectural notion in a technologically flexible manner. Fig. 5 shows an enterprise communicator's implementation of multiple interfaces.



AC	=	Application Connect	MF	=	Message Filter
CU	=	Communicator	MM	=	Message Manager
HI	=	Host Information	OC	=	Object Configurator
II	=	Implementation Information	OP	=	Operational Control
LM	=	Log Manager	PR	=	Principal

Fig. 5. *An enterprise communicator's multiple interfaces.*

Channels. Concert's object-oriented component interfaces are primarily intended to be implemented using CORBA-based or OLE-based technologies. However, certain component capabilities are better suited for other representations that are not necessarily object-oriented or for implementations using technologies other than CORBA or OLE. For example, backwards compatibility with existing standards or stringent performance constraints might dictate the use of other technologies.

In Concert, a component can have interfaces that are not object-oriented. These interfaces are referred to as *channels*. Channels generally do not offer access to the full set of component capabilities that are represented by a component's object-oriented interfaces, but they do provide an architectural basis for representing alternative communication mechanisms.

For example, an enterprise communicator implements TCP/IP channels over which it can send and receive character-formatted HL7 messages. Contemporary applications use a communicator's object-oriented interfaces to send and receive messages, but legacy applications can use a communicator's TCP/IP channels.

Interface Perspectives. During the early development of Concert, most of the emphasis was on the interfaces that represented a component's application capabilities. These interfaces support the ability to use the component to construct healthcare applications.

For example, the application capabilities of an enterprise communicator are represented by the following three interfaces:

- The application connect interface enables an application to connect to and disconnect from a communicator. When connected, an application can send or receive HL7 messages. When disconnected, messages will be buffered for the application until the next time it connects.
- The message manager interface enables an application to create new empty messages that it can fill with data and send and also receive messages that have been sent by other applications.

- The message filter interface enables an application to instruct an enterprise communicator to filter messages based upon their data content. Messages that are filtered are not delivered to the application. For example, an application might only want to receive messages that pertain to a particular patient. The enterprise communicator will send to the application only those messages that pertain to the indicated patient.

It was soon recognized that the application construction interfaces represented only one perspective for defining a component's interfaces and that there were other perspectives that needed to be represented. Specifically, within a healthcare enterprise, there are at least two other perspectives that need to be considered:

- System integration perspective, which is concerned with interconnections within and between systems for the purpose of establishing interoperation (typically based upon relevant standards).
- System management perspective, which is concerned with how systems are configured, monitored, administered, and maintained to preserve desired availability and performance levels.

These perspectives turn out to be extremely important as soon as one starts to address basic issues such as how a component is started or halted, or how data within a component is accessed by systems and applications that are not component-based.

For example, with an enterprise communicator, there are two system integration interfaces. One is an object-oriented interface that enables a communicator to send and receive binary-encoded HL7 messages. The other is a TCP channel that enables a communicator to send and receive ASCII-encoded HL7 messages.

For system management purposes, a communicator supports seven object-oriented interfaces and one SNMP-based channel. The breadth of functionality needed to manage a communicator exceeds the functionality needed to use it for application purposes. While this situation was surprising at first, it is consistent with the notion that enterprise-capable components must be inherently manageable. For example, it would not be practical to deploy communicators throughout an enterprise if there were no way to monitor their performance and intervene from a central location when problems occur.

The concept of organizing a component's interfaces in terms of application construction, system integration, and system management perspectives is one of the cornerstones of Concert. It is this way of thinking about components that has enabled Concert to provide the basis for components that are truly capable of enterprise-wide deployment and use.

In general, the interfaces that make up these three perspectives can be thought of as providing an architectural foundation for component use. Well-defined interfaces organized in a useful way lower the obstacles to using components in a black-box manner to construct systems.

There is, however, a fourth perspective defined in Concert. The component customization perspective represents the concept that a component may have internal interfaces that are similar to traditional application programming interfaces. These interfaces can be used to modify a component's functionality. The important distinction from an architectural perspective is that the customization interfaces offer access to a component's implementation and should not be confused with the external view offered by the interfaces for the other perspectives.

Hardware analogies for software systems are often a stretch, but the following analogy for a Concert component and its various interface perspectives has proven to be effective. A Concert component has sophistication that is roughly analogous to a printed circuit board, such as a sound card that one might plug into a personal computer. The sound card provides application construction interfaces for programs that enable the user to create and control sounds.

The sound card also provides:

- System integration interfaces so that the sound card can be used in conjunction with external MIDI-based instruments (i.e., instruments that support the Musical Instrument Digital Interface) or with an audio speaker
- System management interfaces, often in the form of LEDs that indicate the card's status and DIP switches that enable configuring the card (e.g., setting interrupt vectors or resetting the card's processor)
- Customization interfaces, such as sockets for additional memory chips, which enable changing the functionality of the card, and unlike the card's other interfaces, expose aspects of the card's implementation.

These key component-interface perspectives are illustrated in Fig. 6.

The principles for organizing and defining interfaces for Concert components in terms of these perspectives has proven to be productive and straightforward to implement using both CORBA and OLE automation technology. The work to conceive, specify, and then hone the definition of each interface can be considerable, but the rewards can be substantial.

A well-thought-out and stable set of interface definitions has enabled component design and implementation to proceed at a brisk pace. Further, the interface definitions form a rich basis for an interesting form of reuse referred to as *specification reuse*. Some of the behaviors for new types of components can be reused from the set of interfaces and associated patterns of use that have already been defined.

The use of a common and relatively constrained set of component interfaces across MPG and its partners will enable components to be developed by a single MPG team, by teams in different MPG organizations, and by one or more of MPG's

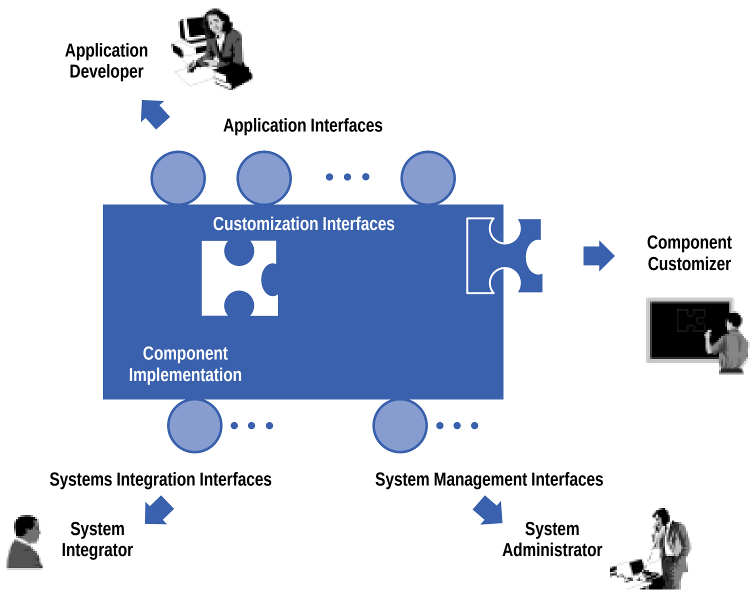


Fig. 6. *Component interface perspectives.*

partners. These interfaces also serve as the basis for open MPG systems. The interface definitions are the key points at which the systems can be opened.

Components and the Architecture Reference Model

The truly important dimension of Concert is not the underlying component model, which is a hybrid of COM and OMA concepts, but the actual components that have been conceived and specified. The first step towards conceiving Concert components was not the development of the component model, but rather the development of a high-level model for healthcare enterprise information systems.

This model, referred to as the MPG architecture reference model (ARM), identifies the key architectural ingredients for healthcare enterprise-capable applications and systems of applications.[5] These ingredients do not prescribe particular system features or technologies. Instead, they organize the architectural content of a software system into ten major groupings.

Each group describes a broad, but nevertheless partitionable, subset of an overall system architecture. The structure of a system is represented by seven facets, shown on the front of the cube shown in Fig. 7. The characteristics of the system that are transitive across all of the facets are illustrated as three horizontal layers that are stacked behind the facets.

The technique for graphically depicting these characteristics as slices was adapted from work on open distributed processing developed by HP's former Network Systems Architecture group.

The alignment of the boxes that represent the facets is important. The facets that represent system features that are most readily perceived by the end user are located towards the top of the illustration. Adjoining facets have significant interrelationships and influences on each other.

An application in the traditional, intuitive sense is also illustrated as a slice, but this slice only cuts through the three inner facets. In an actual system, the software that corresponds to these facets typically implements application-specific behaviors.

In contrast, the four outer facets represent the functional elements of an application system required to relate applications to each other in a coherent and consistent manner. These outer facets also represent the functional elements of an application system needed to relate the overall system to the healthcare enterprise.

The final element of the architecture reference model is the recognition that an application system is designed, developed, implemented, and supported using tools. The degree to which the design, development, implementation, and support activities are productive is a direct function of the degree to which complementary tools are employed.

Further, for each of these activities, the degree to which insightful knowledge of the healthcare enterprise is applied governs the degree to which the resulting application system meets the business needs of the system's supplier and satisfies the requirements of the clinical, operational, and business customers in the healthcare enterprise.

Outer Facets (in Fig. 7). The enterprise communication facet represents the capability for a system to interchange data with other systems in the enterprise based upon relevant healthcare standards. Important elements of this facet are the data formats and communication profiles that make up the interchange standards.

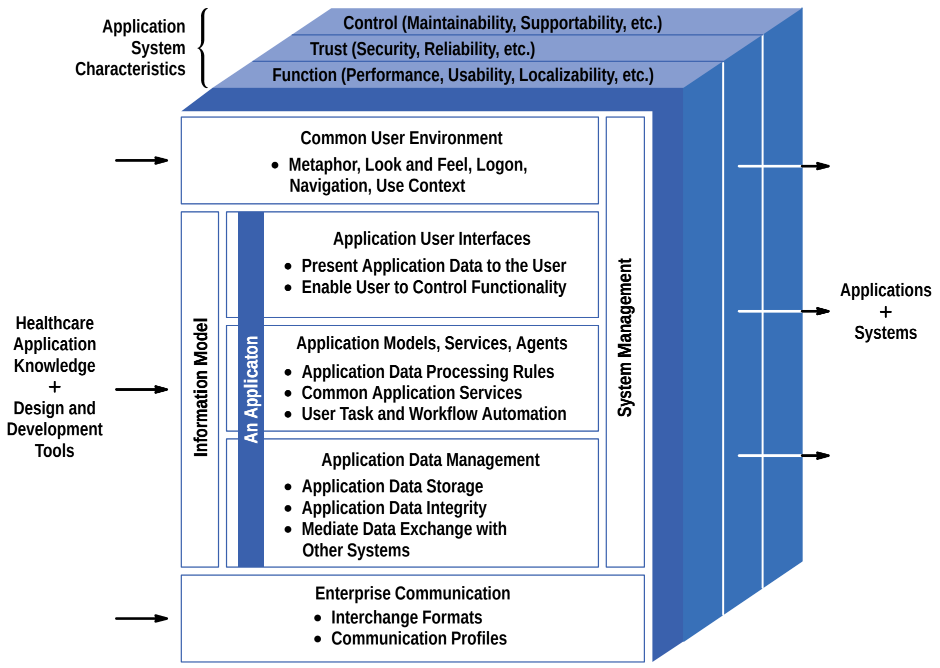

Application System Characteristics
Control (Maintainability, Supportability, etc.)
Trust (Security, Reliability, etc.)
Function (Performance, Usability, Localizability, etc.)
Common User Environment
• Metaphor, Look and Feel, Logon, Navigation, Use Context
Information Model
An Applicaton
Application User Interfaces
• Present Application Data to the User
• Enable User to Control Functionality
Application Models, Services, Agents
• Application Data Processing Rules
• Common Application Services
• User Task and Workflow Automation
Application Data Management
• Application Data Storage
• Application Data Integrity
• Mediate Data Exchange with Other Systems
System Management
Enterprise Communication
• Interchange Formats
• Communication Profiles
Healthcare Application Knowledge + Design and Development Tools
Applications + Systems

Fig. 7. *Concert reference model.*

The information model facet represents the "conceptual glue" that is essential for deploying an application system within an enterprise. The information model identifies and defines the entities and concepts that are important in the domain of the healthcare enterprise. The information model also helps ensure that these entities, concepts, terminology, and clinical processes have a consistent interpretation across all parts of an application system and the enterprise as well as between different but related applications.

The system management facet represents the "operational glue" that enables the uniform and consistent management of the system. This includes capabilities to:

- Turn the system on and off
- Assign passwords for users
- Install new software revisions
- Configure the functionality of the software
- Detect, enunciate, and log faults
- Intervene to correct faults
- Adjust performance parameters and resource utilization levels
- Provide end-user help-desk functionality.

The common user environment facet defines a unifying metaphor that governs user interactions with the underlying applications. For example, for an electronic medical record application system, the metaphor might represent patient data as sections in a virtual three-ring binder.

Under the umbrella of the metaphor, the common user environment also defines the healthcare-specific approach to application user interface look and feel (e.g., clinically appropriate colors, fonts, terminology for common menu selections, etc.), and it provides the highest-level controls, which enable the user to navigate to and between applications.

In addition to these specification-oriented elements, the common user environment includes capabilities that enable the user to log on once to an application system and to establish and manage a *use context* which is applicable to any of the underlying applications. The use context can include settings that identify the user and describe the user's clinical role, characterize the user's physical location, and indicate the user's natural language preference and default preferences for application appearance and control settings.

For example, a physician's use context might include the list of patients that the physician is responsible for. This list resides within the implementation of the common user environment but is accessible to any application in the system.

As the physician switches between applications, applications are provided with information about the patients on the list without requiring the physician to reestablish the list. The continuity provided by the use context enables applications to achieve a high degree of coordination and cooperation. These qualities benefit the physician by providing a simpler and more efficient user interface.

Inner Facets. The outer facets of the architecture reference model define an enterprise environment within which an application participates. An application is described in terms of three basic application facets. Representing an application in this manner makes it possible to factor the responsibilities of an overall application into more granular categories.

While reminiscent of the increasingly popular multitier client/server systems (in which application processing is distributed across a client and a hierarchy of servers), the three application facets are not about client/server computing. Instead, they are about decomposing application software into three distinct sets of responsibilities. This decomposition serves as the basis for scalable and extensible application implementations that can be deployed on a single computer or on a two-tier or N-tier client/server network.

The application user interface facet is responsible for presenting application data to the user and for providing mechanisms that enable the user to interact with and control the application. In this regard, the fundamental role of the user interface is to transform computer-based data into tangible entities that a user can perceive and manipulate. While this is clearly the overall responsibility of an application, the user interface portion of an application is focused on the ergonomic and human-factor aspects of this transformation.

The models, services, and agents facet is responsible for:

- Models
  - Validating user inputs before performing significant application data processing tasks and then performing these tasks
  - Mediating the transformation of application data into concepts and organizations that facilitate populating a user interface with application data
- Services. Providing application-level facilities that are common among but independent of any particular application
- Agents. Automating individual user tasks and multiuser workflows.

The models, services, and agents facet represents a substantial subset of an application's overall responsibilities. However, this facet is notably devoid of any responsibilities pertaining to the direct interaction with the user or with underlying data sources. This facet is neither responsible for the "face" put on the application data, nor is it responsible for the application data. Instead, this facet serves as the bridge in the transformation of data into entities that are tangible to the user.

The application data management facet is responsible for:

- Storing application data that is important to the user and the enterprise
- Mediating the exchange of application data with other systems in the enterprise
- Enforcing the information-model-based rules that ensure the semantic integrity of the application data over time.

This facet is easily confused with a database. However, a database is a particular technology, while application data management represents a set of related responsibilities. For example, application data could be stored in a file or come from a real-time feed (e.g., a patient-connected instrument) as well as from a database.

Further, one of the key responsibilities of this facet is to enforce fundamental data integrity rules (often referred to as business rules). This includes rules based upon the semantics of the data as identified in the information model (e.g., the valid set of operations that can be performed on a medication order) and enforcement of more basic consistency rules (e.g., ensuring that updates that affect multiple data items are reliably performed on all of the data items).

Application System Characteristics. The final part of the architecture reference model describes various characteristics of an enterprise application system that requires the participation of all of the architecture reference model facets.

Functionality is the characterization of an application system in terms of user-perceived qualities that are independent of any one application but must be adequately supported by all applications. These qualities include performance, usability, and localizability.

Trust is the characterization of an application system in terms of its responsibilities to provide users with a system that is secure, reliable, and available when needed.

Control is the characterization of an application system in terms of its capabilities to be administered, managed, supported, and serviced.

Status and Conclusions

Concert was first applied in a deployable prototype electronic medical record (EMR) system that was developed by HP Laboratories and the Mayo Clinic for use at Mayo's Rochester, Minnesota site. Protoypes based upon four types of Concert application components were developed for this project.

The architecture was subsequently applied by MPG to the development of the enterprise communication framework (ECF). An implementation of the enterprise communication framework has been provided to the core members of the Andover Working Group.

For both of these projects CORBA and OLE technologies were employed and development proceeded on HP-UX*and Windows®-NT platforms. Substantial practical experience was obtained, and several important architectural refinements were introduced. Most notably, however, the key concepts described in this paper were exercised and validated.

More recently, Concert has served as the basis for a variety of information system product development activities within MPG. The specifications, experiences, and some of the software developed for the EMR and the ECF are being applied. It typically takes an object-oriented software developer about two weeks to become familiar enough with the architecture to begin productive development of Concert-based software. Indications are that once this investment is made, the specifications provide a solid, self-consistent basis for system development.

The next challenge is to further optimize development productivity through the creation of Concert component development frameworks. These frameworks would provide code skeletons for partially implemented components. Armed with an appropriate set of productivity tools, application developers would be able to add the necessary features to the skeletons to create fully functional components. Tools would also help the developer “wire” the components together to form an application or a system of applications.

Acknowledgments

A substantial number of people were involved in the conceptualization and specification of Concert. It is through the efforts of all of these people, working together in concert, that we were able to develop a comprehensive architecture in a relatively short amount of time. The participants include: Don Goodnature, Mike Higgins, Jeff Perry, Jaap Suermondt, and Charles Young from HP Laboratories Analytical and Medical Lab, Rafi Ahmed, Philippe De Smedt, Louis Goldstein, Jon Gustafson, Pierre Huyn, Joe Martinka, James Navarro, Tracy Seinknecht, and Joe Sventek from HP Laboratories Software Technology Lab, Tom Bartz and Dean Thompson from the HP Network and Systems Management Division, Robin Fletcher, David Fusari, Jack Harrington, Nico Haenisch, and Geoff Pascoe from MPG R&D, Bob Anders, Steve Fiedler, Peter Kelley, Anthony Nowlan, and Mike Stern from MPG HealthCare Information Management Division R&D, Rick Martin and John Silva from the MPG Customer Services Division, and Alfred Anderson, Pat Cahill, Calvin Beebe, Woody Beeler, Tom Fisk, and Bruce Kaskubar from the Mayo Clinic. In addition, the author would like to thank Mark Halloran, HP MPG R&D manager, for his enthusiastic support and executive sponsorship of this work.

References

1. *Common Object Request Broker: Architecture and Specification, Revision 1.2*, Object Management Group, 1993.
2. K. Brockschmidt, *Inside OLE, Second Edition*, Microsoft Press, 1995.
3. *Version 2.2, Final Standard, Health Level Seven*, December 1, 1994.
4. *Concert Component Architecture: Component Concepts and Base Specification, Version 1.0*, Concert Document 95-11-1, Rev. A, November, 1995.
5. *MPG Architecture Reference Model, Version 1.0, Rev. A*, MPG Architecture Document 95-9-3, last revised September 21, 1995.

▶ Go to Subarticle 11a

▶ Go to Subarticle 11b

▶ Go to Subarticle 11c

▶ Go to Next Article

▶ Go to Journal Home Page