HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES

HICSS-34 January 3-6, 2001
Outrigger Wailea Resort Maui

Business Object and Component Architectures: Enterprise Application Integration Encounters Complex Adaptive Systems

Jeff Sutherland, Chief Technology Officer, PatientKeeper, Inc.
jeff.sutherland@computer.org, http://jeffsutherland.com/

Abstract: Concepts in Complex Adaptive Systems (*cas*) research are relevant to the development of enterprise business objects and component systems. Many mathematical and computing models have been developed for *cas* in recent years(Cowan, Pines et al. 1994) and much of this work can be applied conceptually to Business Object and Component Architectures now emerging as the mechanism of choice for building large distributed object systems. Holland creates a synthesis of seven basic *cas* concepts, four properties (aggregation, nonlinearity, flows, diversity) and three mechanisms (tags, internal models, building blocks) (Holland 1995). These concepts can organize our discussion of business object systems and inform our understanding of Enterprise Application Integration (EAI).

Several examples of innovative approaches to EAI are provided: (1) an implementation of workflow, XML, and intelligent adapters to integrate disparate front and backend systems in an insurance company merger (Van den Enden, Van Hoeymissen et al. 2000) , (2) mobile/wireless device platform support for integration of large numbers of legacy systems in healthcare, and (3) a cutting edge implementation of "Big Workflow", a distributed internet workflow system for driving all applications in an enterprise via HTTP/XML/SOAP messaging (Sutherland and Alpert 1999). Directions for future research are outlined in an architecture for intelligent agent-based systems that provide goal directed behavior actionable through driving "Big Workflow" enabled, distributed, heterogeneous systems (Maamar and Sutherland 2000). The incrementally enhanced power of these example systems is due to more extensive application of *cas* concepts for each implementation.

Background

Holland defines *cas* as systems composed of interacting agents which respond to stimuli. Stimulus-response behavior can be defined in terms of rules. Agents adapt by changing their rules as experience accumulates and can be aggregated into meta-agents whose behavior may be emergent, i.e. not determinable by analysis of lower level agents (Holland 1995)

Business entities are good examples of complex adaptive systems. The modification time of a business firm is on the order of months or years, about the same amount of time required to enhance its computing systems. Automating business processes renders those parts of the business in software, thus enterprise software systems are examples of *cas*. A business system

has a severely constrained rule set, compared to a typical *cas* system, like New York City or the U.S. economy. At the same time, flexibility, adaptability, and reusability of these systems can determine the ability of an enterprise to evolve and survive in the marketplace. Typical characteristics of *cas* systems are lacking in most business software and business systems could be significantly more flexible if architected with *cas* concepts.

New discoveries in object technology parallel Holland's analysis of *cas.* Event driven, distributed object component systems are "interactive systems." Wegner has shown that interactive systems are not Turing machines (Wegner 1995). All interactions in these systems cannot be anticipated because behavior emerges from interaction of system components with the external environment. Such systems can never be fully tested, nor can they be fully specified. These contemporary, event driven, business object component systems exhibit emergent behavior, a fundamental feature of *cas*. And they are clearly complex, to the extent that the running system may be the shortest description of its behavior. This is certainly true of large enterprise systems running in multiple sites with different custom code and hardware platforms at every site. Supply chain integration of these disparate systems via an intranet, extranet, and/or the internet is often a global EAI development task.

For some years, the author has described a Business Object Component Framework using the diagram below (Sutherland 1998). Objects are aggregated into components. Components are aggregated into meta-components and a hierarchical structure is built to support integration of enterprise software systems. Holland uses an equivalent diagram to describe a complex adaptive system made of of adaptive agents (see Figure 1.1, (Holland 1995), p. 6). Work on Business Object Component systems is evolving along the lines of *cas*, because successful systems are, in fact, instances of complex adaptive systems. Of the 80% of all software projects deemed failures, reasons for failure are often related to the inability of systems to rapidly adapt to changing business needs (Brown 1998). In fact, the failure to design software systems to support *cas* behaviors dooms the majority of them to failure and the minority survivors to premature obsolescence because of their inability to adapt through integration with newer system developments.

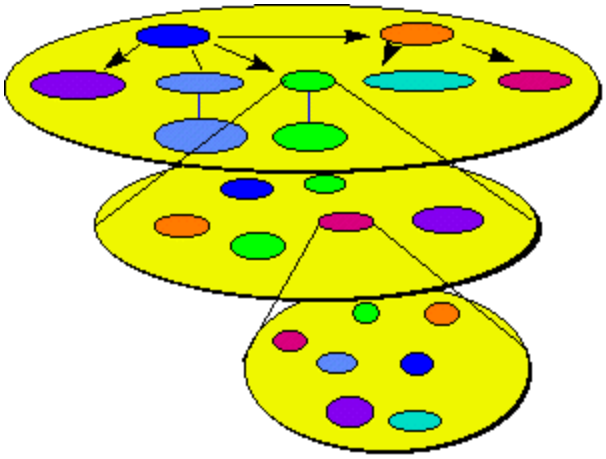

Figure 1: Business Object Component Framework

Enterprise application integration from a Business Object Component/*cas* perspective requires an understanding of Holland's synthesis of *cas* concepts:

1. Aggregation (property) - there are two important modes of aggregation in *cas* systems. Aggregation is a basic mechanism in object modeling and is the basis for identity, a fundamental object concept. Forming components out of objects and enterprise systems from components is higher level aggregation. More important are emergent properties such as intelligence that evolve out of dumb subsystems. This is the basic concept in Minsky's "Science of Mind" (Minsky 1988) or Hofstader's analysis of an ant colony (Hofstadter 1979). Meta-agents (an enterprise) are formed of aggregates of agents (enterprise systems) and exhibit emergent behaviors (revenue, profitability, and cash flow, the indices of value creation).

2. Tagging (mechanism) - this mechanism facilitates the forming of aggregates, from HTML pages to the mechanisms in CORBA or DCOM that allow inter object communication. They facilitate selective mating, i.e. firewalls block certain tagged elements to protect the enterprise. Thus they preserve boundaries between aggregates. They allow us to componentize object models and enable filtering, specialization, and cooperation. They are the mechanism behind the development of hierarchical aggregates that exhibit emergent behaviors like an operating system. The basic mechanisms of evoking operations through messages in object technology are based on tagging strategies.

3. Nonlinearity (property) - nonlinear systems exhibit catastrophic and chaotic behaviors. Traffic flow on the Internet is nonlinear, leading to predictions of the collapse of the network. Brownouts, system loadings, scalability effects are often nonlinear. The arrival, proliferation, and destruction of viruses on the Internet is a nonlinear phenomenon that can be modeled like predator/prey interactions in biological systems. Even more basic phenomenon like revenue prediction in an enterprise financial system has nonlinear behaviors that are often not recognized. The rate of construction of software itself is a nonlinear phenomenon.

4. Flows (property) - workflows are examples of flows in action. Message routing is a flow. Tags condition flows which often exhibit nonlinear behaviors and emergent behaviors. Flows typically have a multiplier effect. Money injected into the economy has an effect out of proportion to the amount, similar to email or other message flows on a network. The recycling effect of flows enables the rain forest, as well as an enterprise computer ecosystem. Individual pieces evolve, die, are replaced or reused, constantly changing the characteristics of the enterprise. Living software is software that is constantly changing due to flows, as rivers change their course. Dead software is eventually detritus that is expelled from the enterprise organism.

5. Diversity (property) - persistence of an individual agent depends on the ecosystem of agents that surround it, whether the agent is an ant in the rain forest or a business object

in an accounting system. The evolution of these agents as software changes causes convergence of system architectures. This is the basis of emergent patterns that reappear again and again in widely disparate environments. It is difficult to evolve a single agent to make it more useful in an isolated context. Usefulness in business object systems arises from interactions between diverse agents as in human societies.

6. Internal models (mechanism) - the utility of complex systems is enhanced if the system can learn from experience and adapt its behavior. The ability of the system to develop and act on internal models that simplify the external world is basic to this mechanism. It allows the system to infer the results of actions before they are taken, and to choose actions that have productive results. The prospects for longevity of software systems depend on this capability, just as in living systems.

7. Building blocks (mechanism) - reuse is dependent on building blocks used over and over again. It is the basis of Moore's law in hardware production. It could be the basis of dramatic improvements in software productivity. Building blocks are the basis for generation of internal models and are essential to the construction of adaptive enterprise systems.

Holland, in his Ulm lectures at the Santa Fe Institute creates a synthesis of these seven basic *cas* concepts, four properties (aggregation, nonlinearity, flows, diversity) and three mechanisms (tags, internal models, buildings blocks) (Holland 1995). These was the theme for the OOPSLA'98 Business Object Design and Implementation Workshop (Patel, Sutherland et al. 1998) and these concepts can organize our discussion of enterprise application integration of Business Object and Component systems.

Enterprise Application Integration as the Art of "Capturing Software"

> It is often taken for granted that as systems evolve over time they tend to become more complex... In single systems it may grow by increases in structural sophistication: the system steadily cumulates increasing numbers of subsystems or subfunctions or subparts to break through performance limitations, or to enhance its range of operation, or to handle exceptional circumstances. Or, it may suddenly increase by "capturing software": the system captures simpler elements and learns to "program" these as "software" to be used to its own ends... A particularly telling example of capturing software is the way in which sophisticated derivatives have arisen and are used in recent years in financial markets... (Arthur 1994)

In recent years, there has been an increasing focus on EAI by industries both producing and using software. Time and resources are no longer available to rebuild legacy systems as they become more complex and are required to change at a more rapid pace. They must be "captured" like a cowboy lassos a wild calf. The animal must be hog-tied and branded, and driven to market with the herd.

Example 1: "Capturing" Software in an Insurance Company Merger

The Business Object and Component Workshop has been held annually at the Object-Oriented

Programming, Systems, Languages, and Applications (OOPSLA) conference since 1995. It has been a forum for an ongoing discussion between leading software developers from industry, academia, and government from most of the leading countries in the world.

At OOPSLA 2000, Van den Enden, Hoeymissen, Neven, and Verbaeten (Van den Enden, Van Hoeymissen et al. 2000) provided a creative example of "capturing" two disparate enterprise systems. They needed to integrate these systems because of the merger of two insurance companies. The front office system from the first company was built on Sun hardware using the Forte 4GL environment. The second company had a backend system that ran on Unisys mainframe hardware and was coded in LINC, a Unisys code generation language.

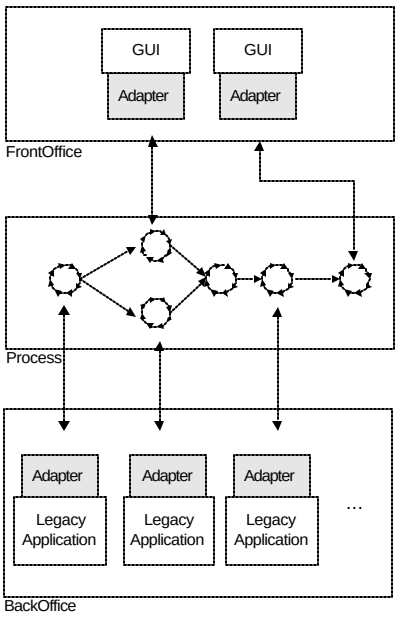

Figure 2: Integrating two disparate systems (Van den Enden, Van Hoeymissen et al. 2000)

Van den Enden et. al., of Business Integration Company and DistriNet Research Group in Belgium, decided to use a workflow engine, intelligent adapters, and XML messaging (Bosak 1997) as the core of their integration strategy. New technology was superimposed on the old in order to enable (1) interaction between web HTML clients and mobile WML clients, (2) utilization of standard transform mechanisms with XSL/XSLT, (3) easy integration with future systems in electronic marketplaces through XML, and (4) validation and language mapping capabilities available with XML DTD and XML Schema tools.

Van den Enden's architecture "captured" the disparate systems with a workflow engine (Sun's Forte Conductor). All business logic is encapsulated in a workflow, and the architecture uses intelligent adapters to provide for the 'glue' that links the external applications to the workflow. Adapters transform XML message formats into other data formats or into objects and are able to take different actions based on the content of the message.

Sun's Forte Conductor graphical tools are used to set up a process definition which describes

how to initiate a process, what step the process is in, and who is responsible for executing the step (the front end user or the backend system). All details of the frontend or backend systems are hidden from the workflow layer. Thus Forte Conductor "conducts" the execution of a process instance by human and machine interactions.

The backend architecture of this system is described in the Figure below. When a node in the process instance requires the cooperation of the backend system, it sends an XML message to a "robotic" client. This agent orchestrates the flow of commands required to integrate the backend system into the workflow.

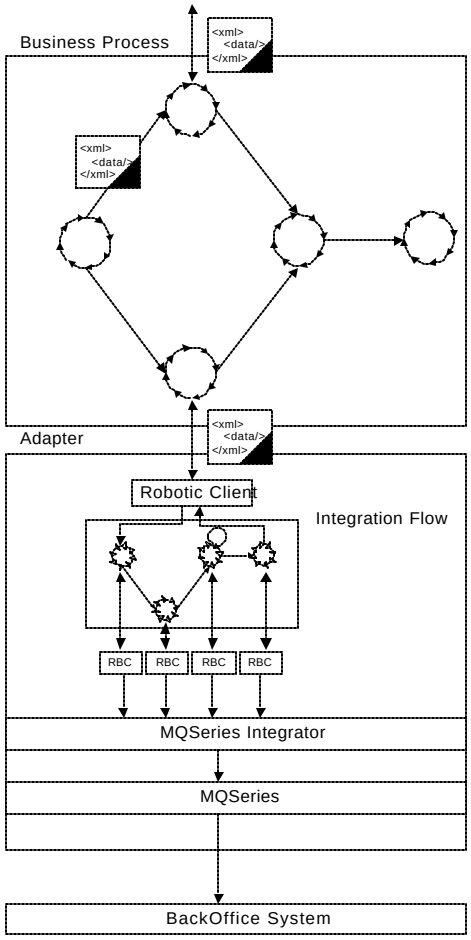

Figure 3: Overview of the backoffice adapter (Van den Enden, Van Hoeymissen et al. 2000)

Similarly, the Conductor workflow engine sends XML messages to a frontoffice adapter which insulates the frontoffice system from the workflow layer as in the next Figure.

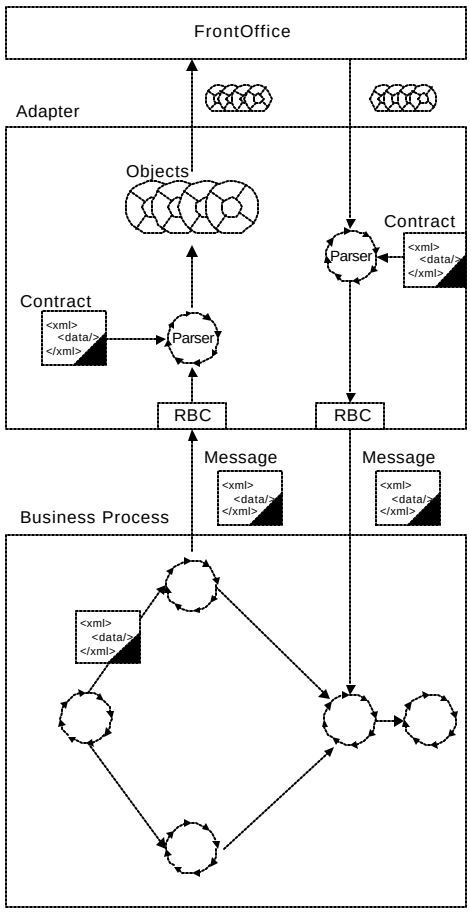

Figure 4: Overview of the frontoffice adapter (Van den Enden, Van Hoeymissen et al. 2000)

In this architecture, two legacy systems are **aggregated** into a single system by adapters controlled by the workflow engine. **Nonlinear behavior** induced by actions of goal seeking agents is avoided in this system by essentially hardcoding workflows and tuning them prior to production. **Flows** are managed by the workflow engine and routed using **tags** supported by the infrastructure of XML. and **diversity** is managed by shielding the workflow engine with "robotic" clients. **Internal models** are essentially hardcoded. An **building blocks** used in **aggregation** are enabled by workflows interacting with adaptors that manipulate legacy systems.

We would not expect emergent behavior in this system because of hardcoding workflows and implicitly defined and fixed internal models. However, since the architecture abstracts business processes from the more rigid legacy system, it would support future extension of the system to support agents with adaptive internal models that could dynamically define workflows within the limits of defined adapters. In particular, this system implements a fundamental concept that will be characteristic of future adapative systems – automated workflow drives business processes, rather than human interaction. As a result this is an excellent approach to "capturing" the software of two legacy systems. It preserves the legacy investment while freeing the legacy systems from many limitations.

Example 2: Mobilizing Healthcare Information to Save Time, Lives, and Money

> The Internet explosion has inspired an outpouring of predictions that the health sector's long-awaited breakthrough in information management is finally at hand. But will network computing really help create order amid the befuddling maze of insurance claims, clinical records, and quality data in which the key to a more efficient system now lies hidden? Or will the ultimately localized, idiosyncratic, and fragmented enterprise of care continue to prove resistant to rationalization? (Inglehart 2000)

A more sophisticated problem of enterprise integration is introducing "point of sale" technologies into healthcare. Most people are unaware that the United States is tied with Croatia for the lowest level of automation at the point of care of any country on the planet. Over 95% of clinicians have no automation at the point of care in the United States versus more than 90% with some level of automation at the point of care in the United Kingdom and Canada.

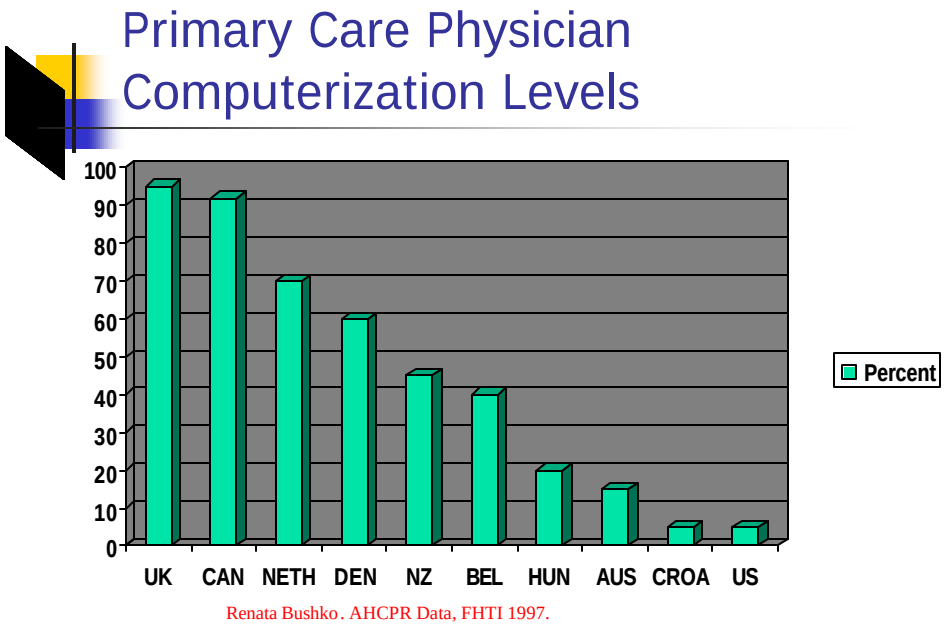

Figure 5

The net result of this is a tremendous loss of money due to lost or incorrectly code charges (an average of 8-10% in large Integrated Healthcare Delivery Networks). Loss of funds is even greater in reimbursement for laboratory tests due to miscoding and failure to demonstrate medical necessity requirement for insurance reimbursement.

Even worse is the fact that medical error is (conservatively) the fourth leading cause of death in the United States and that minimal levels of automation would reduce these deaths by 50-80%.

This has been declared a national emergency by the National Academy Insitute of Medicine. (Kohn, Corrigan et al. 2000)

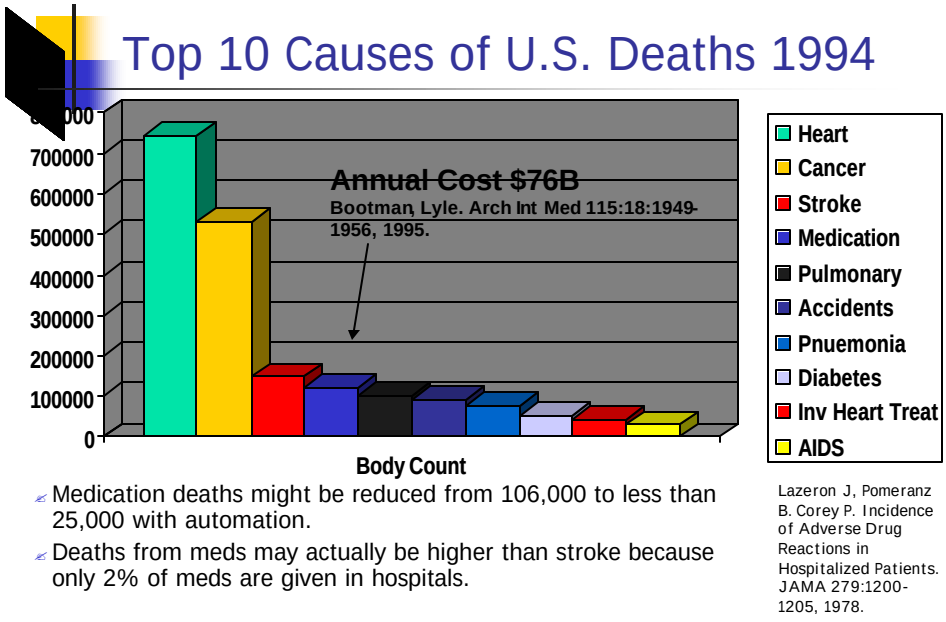

Figure 6

Solving this problem is a complex enterprise application integration effort. First, a large Integrated Delivery Network (IDN) will often have hundreds of disparate software systems, several of which will need to be integrated to support even a single limited application like generating a bill for treatment. Second, physicians are inherently mobile and have no immediate access to any of the information in these hundreds of systems. Adoption of personal computers and online medical records is vanishingly small at the point of care. However, almost 30% of physicians today carry a mobile device such as a Palm Pilot and the number is growing quickly due to physician interest. Third, the cost of mobile devices and wireless communication is dropping rapidly to the point where return on investment is high, not to say astronomical if the cost of unnecessary death and disability is factored into the equation. One could argue that reducing the fourth leading cause of death by 50-80% is the most humane act that computer professionals could possibly provide in terms of reduction of human pain and suffering.

Tackling this problem requires four layers of distributed systems technologies. Cache consistency across all layers of these systems must be maintained precisely to avoid medical error (Franklin, Carey et al. 1997; Lee, Hwang et al. 1999). An integration platform architecture for support of mobile/wireless applications at the point of care is shown below.

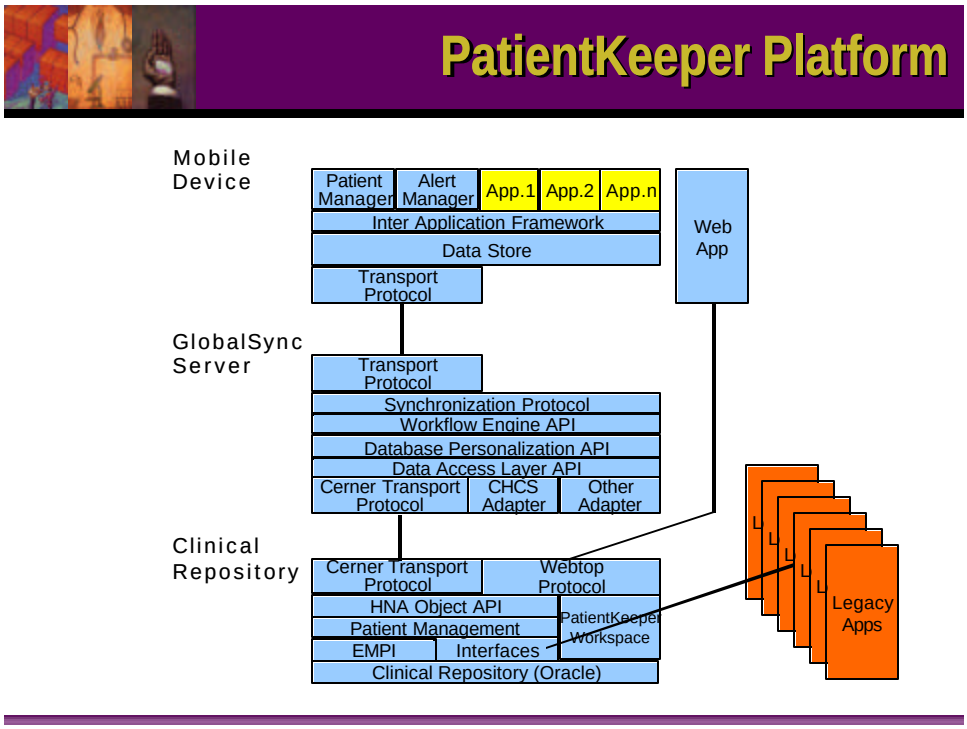

Figure 7

The top layer of this architecture is the mobile device. While the platform needs to support a wide variety of device types, an individual clinical only wants one device in their coat pocket. An application API on the mobile device is essential to allow the device to support a wide variety of rapidly proliferation applications.

The second layer of the architecture is a synchronization server which must support a wide variety of device types. Information flowing to the mobile device must be personalized to the specific device, clinician, and application. For example, if Dr. Palm is a cardiologist, he has a set of applications oriented towards cardiology. He wants to see his own patients on his Palm Pilot and their lab results. If he is in his office, he may want to be alerted on his Palm Pilot. When he is on the golf course he may want to be paged. If he is on call he will want another physicians patients to transparently appear on his mobile device. The synchronization server must stage data, manage personalization, and handle routing, alerting, and messaging. It must keep the data cache on the mobile device synchronized with lower level databases in the architecture.

In order to manage a complex set of clinical and financial data requiring extensive authorization, security, and auditing features, a robust, fault-tolerant, clinical repository is required. This is the third layer of the architecture. In addition to managing patient data, the repository must manage multiple record numbers for the same patient which exist independently on dozens of legacy systems. It must also manage a complex network of enterprise application integration varying between dozens of different applications. For example, if Dr. Palm is in the hospital on rounds he wants data to flow to and from the hospital financial and clinical system. When he goes back to his ambulatory clinic, he wants to see data flowing to and from his clinic financial and clinical

systems. He wants all this to be transparent to whichever hospital or clinic he is in at the moment. He wants data seemlessly available anywhere or he will not use the device.

The clinical repository in this example is integrated using a component object strategy. The repository provides object-oriented component interfaces to the synchronization server. The messaging protocol between the synchronization server and the repository is XML using SOAP (Nielson 2000) as an RPC mechanism. The critical need for a component architecture to provide seamless integration is discussed in the next section.

The fourth layer of the architecture is the legacy systems themselves. They can be integrated via adapaters as in the previous insurance example, or interfaced using standard message formats or proprietary legacy interfaces which can be different for each legacy system.

The architecture **aggregates** multiple heterogeneous systems through component objects, adapters, or messaging interfaces. It also **aggregates** data from backend legacy systems and manages the consistency of data across layers of the architecture. **Flows** are handled by workflow engines at both the synchronization server and clinical repository layers of the architecture. **Tagging** is supported by XML infrastructure. **Diversity** is shielded from the mobile device by the clinical repository. **Building blocks** are **aggregated** with a wide variety of integration patterns and mechanisms for interoperability. The system is complex enough to induce **nonlinear behavior** but this must be managed by manual tuning or coding. **Internal models** are implicitly defined by hardwiring components for specific applications. There may be limited goal seeking behavior induced by hardcoded business rules in various system layers.

A unique feature of this architecture is that mobile devices can be used as a remote control to drive the enterprise. Workflow drives business processes. However, humans at the remote control serve as intelligent agents with goal-seeking behaviors.

This architecture is an excellent attempt to handle complexity but is not adaptive. New configurations must be manually created. To raise the level of adaptability requires moving a workflow engine to the center of the architecture while simultaneously distributing it across all computing platforms in the enterprise. This is the focus of the third example, but first some comments on component architectures essential to the workflow strategy.

Creating flexible, adaptable, and reusable software requires a component model

As business models are renewed, software architectures must be transformed. A Business Object Component Architecture (BOCA) is an effective solution for dynamic automation of a rapidly evolving business environment. Dynamic change requires reuse of chunks of business functionality (as noted by Arthur in his description of the *cas* "software capture" effect). A BOCA must support reusable, plug compatible business components. The two primary strategies now being used for implementing client/server systems to support reengineering of business processes are visual 4th Generation Languages and classical object technology. While both of these approaches are recent improvements in system implementation, neither of them effectively implement plug and play Business Object Components.

A group of objects is the ideal unit of reuse. Groups of objects behave as a higher level business process and need a clearly specified business language interface. Proponents of *cas* should note carefully Abelson and Sussman's comments in their classic MIT computer science text (Abelson, Sussman et al. 1996). The power of computing lies in recursively writing higher levels of language that are supported by lower level languages (thus inducing emergent behaviors).

Business Object Components need to be encapsulated with a protocol that allows efficient communication with other objects on the network. Work on the concept of Ensembles has shown that there is a minimal design specification for a plug compatible component (Love 1993; Sutherland and McKenna 1993). A business object component needs semantics that extend beyond the syntax of COM components and JavaBeans containers.

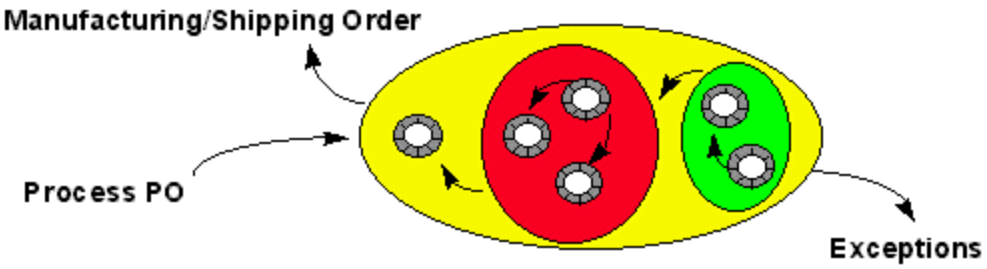

Figure 8: Business Object Component

Consider a typical client/server application like an order entry system. This core component of this system takes a Purchase Order as input and produces a validated order as output or, alternatively, raises exception conditions. Key issues are (1) internals of this component should be a black box to the external world, (2) externals of this component are a black box to internal objects, (3) external components can register for notification of interesting internal events, (4) internal components can be changed or upgraded without affecting external operations, (4) external components can be changed without affecting the component's operations, and (5) any internal component can be run on any processor and the whole component retains its functional integrity. These objectives are largely achieved by emerging COM+ and Enterprise JavaBeans specifications at the syntax level.

More importantly from a business perspective, the Business Object Component must guarantee proper behavior in response to a set of scenarios that ultimately trace back to use cases. These scenarios must be evoked by messages in a standardized context, including timing and sequencing, to guarantee proper operation of the component in any system that uses the component according to its design specification. While these minimal requirements were worked out years ago by Sutherland and McKenna (Sutherland and McKenna 1993) and pursued by the OMG Business Object Domain Task Force since 1995, they have yet to evolve into any standard approach for building plug compatible business components.

It is critical that standard Business Object Components be available to enable a business system to function as a complex adaptive system. Adaptive means that systems must easily evolve and this requires continuous changing, updating, adding, subtracting, and rearranging components. When changing one component causes ripple effects throughout a system, the cost of rewriting

major portions of the system slows evolution to a snails pace. This is the state of the 80% of business systems that are viewed as unsuccessful implementations in industry today (Brown 1998).

Example 3: "Big Workflow" Drives the Enterprise

"Big Workflow" is a term that has been used to describe workflow that crosses multiple applications and multiple vendors to support patient flow across an healthcare Integrated Delivery Network (IDN). Similar systems have been implemented or are under construction in manufacturing and other vertical application domains. As an example, HealthSystems Minnesota has carefully defined their business processes for managing patient flow across multiple institutions within their IDN.

- Workflow must be managed across over 300 large applications provided by dozens of vendors.
- They would like the capability to superimpose HealthSystems business processes across all vendor applications.
- They would like to modify their business processes globally without modifying vendor systems.

Sutherland and Alpert presented an implementation of "Big Workflow" at the OOPSLA'99 Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems II and the OOPSLA'99 Business Object and Component Design Workshop (Sutherland and Alpert 1999). This work was initiated based on an OOPSLA'97 Business Object Workshop presentation by Santanu, "Essential Requirements for a Workflow Standard" (Paul, Park et al. 1998). Paul reviewed work on "Rainman: A Workflow System for the Internet" at IBM T.J. Watson Research Center and pointed out deficiencies in the WfMC architecture (Paul, Park et al. 1997). When Paul left IBM in 1998, he served as a consultant on the "Big Workflow" project in order to help take workflow design the next step beyond "Rainman."

"Big Workflow" was implemented based on several fundamental requirements. Any server on the Internet can host any component of the system (process manager, workflow engine, observers, worklists). W3C standards (HTTP/XML/SOAP/LDAP, see http://www.w3c.org/xml and (Howes, Smith et al. 1999)) was used exclusively in the implementation. There was no single point of failure, and any system that supports an XML remote procedure call (SOAP) could participate in the workflow.

Workflow can be viewed as a primary component in architecting applications where flow of control is abstracted out of application business logic. The application model layer (see figure below) becomes a set of services that can be manipulated by a workflow engine. Alternatively, a legacy system can expose a set of services that can be driven by an external workflow engine. If the user can create and alter process definitions (workflow protocols), a process definition can serve as a simple agent which drives business processes.

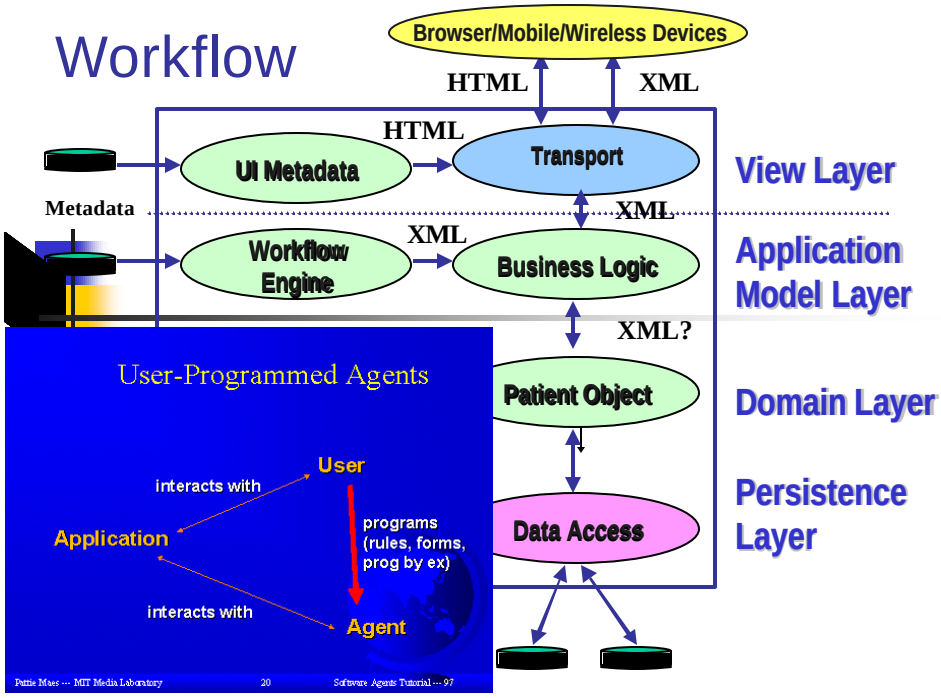

Figure 9: Application Architecture for Workflow

The workflow system is composed of a process manager that receives an event. The process manager then looks in the LDAP for a URL for the appropriate process template, policy manager, and workflow engine. The process manager using the workflow engine to determine the next step in the process, queries the policy manager to see who is responsible for that step (human or machine), queries the LDAP for the URL of the worklist for the responsible party, and posts the work item on the appropriate worklist somewhere on the Internet (see figure below).

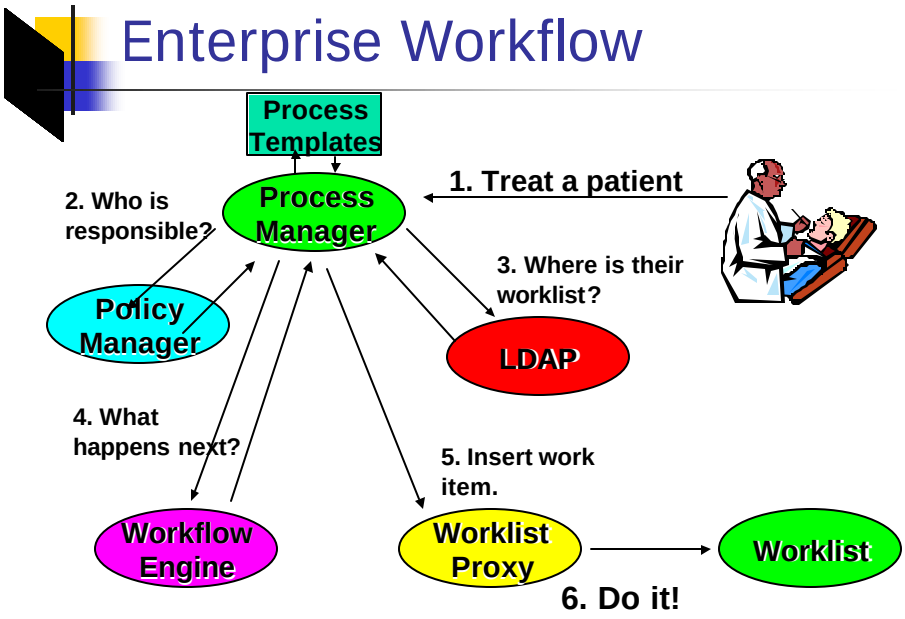

Figure 10: Internet Workflow for the Enterprise

A more detailed description of the process of creating a work item is described in the figure below.

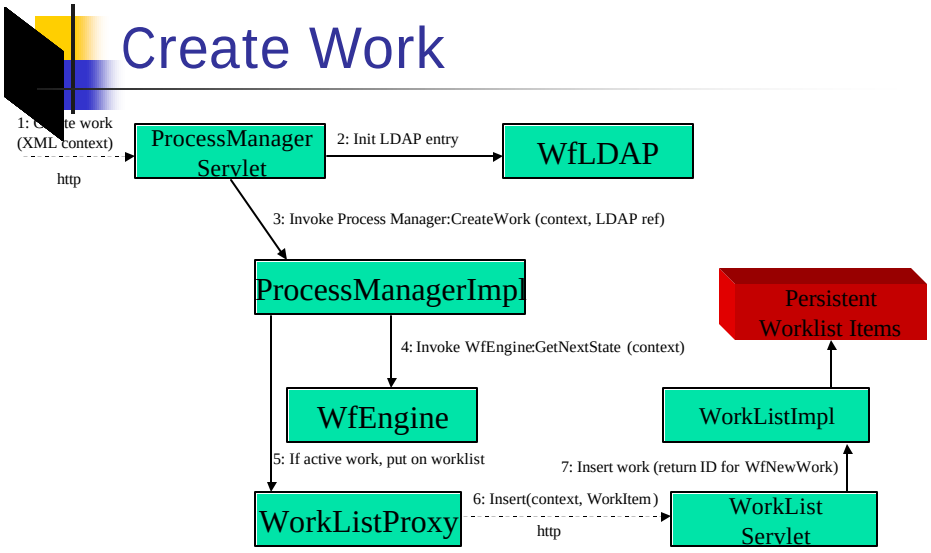

Figure 11: Distributed, Scalable, Fault-tolerant, Reroutable Workflow

The system can set up multiple Observers to view the state of any workflow anywhere on the Internet. A Worklist will notify participating Observers upon any change in state. A distinguishing feature of this implementation is that all workflow context is distributed to worklists. There is no central point of control. Any Workflow Engine on an server on the Internet can drive the workflow process. If an Observer notices that a workflow process is not completing in an appropriate period of time, the Observer can delegate this work item to another server. Thus the system as a whole is fault tolerant and scalable with the capability of automatically routing around bottlenecks.

This system was the first fully distributed Internet workflow implementation that was sufficiently scalable, fault-tolerant, and about to reroute around bottlenecks or failures known to the OOPSLA'99 Workflow Workshop participants. In the view of IBM T.J. Watson Laboratory scientists, it transcended anything created at IBM. Any application on the Internet that exposed a set of services with an XML/SOAP interface could participate in any workflow. Such applications could be automated to interoperate on a global scale. This was viewed as a basic platform on which to build the next higher layer of an architecture that could support goal-seeking *cas* behaviors. This is a major thrust of future research directions.

EAI Research Directions

Interactive, autonomous, business object components could evolve independently with a full implementation of an agent-based enterprise system. Some of these components could be intelligent agents roaming the net to perform complex tasks based on enterprise workflows. Workflows in these systems must be managed in a more sophisticated way than current Workflow Coalition or the Object Management Group specifications prescribe (Schmidt 1998). They will exhibit complex behaviors, catastrophic events, and chaotic interactions, all phenomena subsumed under the umbrella of "complex adaptive systems (*cas*)." They are under intensive research for use in predictive economic models (let the computer beat the stock market, deploy new derivatives, or perform arbitrage), the building of artificial life forms for the analysis of biological systems, computer models that can independently adapt and evolve, "avatars" that can personally represent the creator in an Internet chat room, to note only a few examples.

The previous examples had static implementations that did not allow the system to adapt its operations based on the dynamic state of the runtime environment. Next generation systems will allow business object components to decide with whom to collaborate, what services to offer, what services to request, and what visible behaviors to exhibit (Maamar and Sutherland 2000).

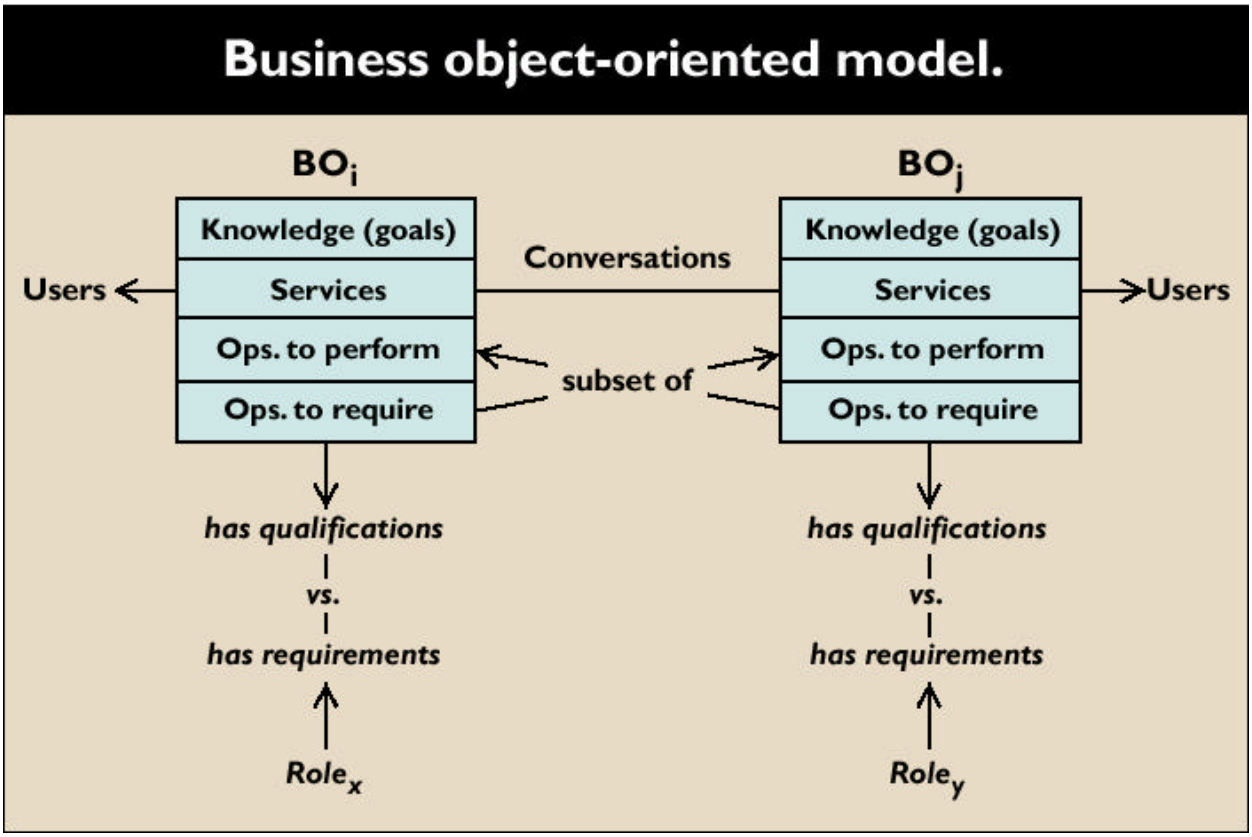

Figure 11 from (Maamar and Sutherland 2000)

In the Figure above, business object (BO) components have a multilayered architecture. The first layer contain knowledge about goals to seek. The second layer advertises services to other business object components that may be performed by this BO and are associated with the goals of the BO. The third layer provides the operations used to perform the service and these operations may be packaged into workflows. The fourth layer contains references to operations performed by other BOs.

Coordination between BOs is carried out by "conversations" based on well-known coordination strategies (Jennings, Sycara et al. 1998). A Business Object Communication Language (BOCL) needs to be standardized as part of a Business Object Component Architecture (BOCA) to provide consistent interoperablility between business object component systems.

Assuming that operations in a BO are packaged into workflows that can be driven automatically by a "Big Workflow" implementation, a BO could dynamically **aggregate** BO **building blocks** to execute goal seeking behavior.

Conclusion

Engineers working on Business Object Component Architectures and EAI would benefit by gaining an understanding of *cas* concepts and applying them to their work. These concepts are larger in scope and broader in application than some of the Business Object work now under

development. Some of the best logical and mathematical minds of our time are gravitating to *cas* research. We can expect fundamental breakthroughs in our understanding of how to build complex adaptive systems that could translate directly to Business Object Component Architectures if the dialogue and definitions used by Business Object specialists aligned with *cas* research. Furthermore, the Business Object community could provide some of the best testing ground for *cas* concepts and fill in the major gap in *cas* research by providing hard data on production EAI systems exhibiting *cas* qualities.

References

Abelson, H., G. J. Sussman, et al. (1996). Structure and interpretation of computer programs. Cambridge, Mass.; New York, MIT Press ; McGraw-Hill.

Arthur, W. B. (1994). On the Evolution of Complexity. Complexity: Metaphors, Models, and Reality. Proceedings Volume XIX, Sante Fe Institute Studies in the Science of Complexity. G. A. Cowan, D. Pines and D. Meltzer, Addison-Wesley.

Bosak, J. (1997). XML, Java, and the future of the Web. Sun Microsystems.

Brown, W. J. (1998). AntiPatterns: refactoring software, architectures, and projects in crisis. New York, Wiley.

Cowan, G. A., D. Pines, et al. (1994). Complexity : metaphors, models, and reality. Reading, Mass., Addison-Wesley.

Franklin, M. J., M. J. Carey, et al. (1997). "Transactional Client-Server Cache Consistency: Alternatives and Performance." ACM Transactions on Database Systems **22**(3): 315-363.

Hofstadter, D. R. (1979). Gödel, Escher, Bach : an eternal golden braid. New York, Basic Books.

Holland, J. H. (1995). Hidden order : how adaptation builds complexity. Reading, Mass., Addison-Wesley.

Howes, T., M. C. Smith, et al. (1999). Understanding and Deploying Ldap Directory Services, MacMillan Technical Publishing.

Inglehart, J. K. (2000). "The Internet and Health: Vision." Health Affairs **19**(5): 8.

Jennings, N., K. Sycara, et al. (1998). "A roadmap of agent research and development." Autonomous Agents and Multi-Agent Systems **1**(1): 7-38.

Kohn, L. T., J. Corrigan, et al. (2000). To err is human : building a safer health system. Washington, D.C., National Academy Press.

Lee, S., C.-S. Hwang, et al. (1999). Supporting transactional cache consistency in mobile database applications. Proceedings of the ACM international workshop on Data engineering for wireless and mobile access, Seattle, ACM.

Love, T. (1993). Object Lessons: Lessons in Object-Oriented Development Projects. New York, SIGS Publications.

Maamar, Z. and J. Sutherland (2000). "Toward Intelligent Business Objects: Focusing on techniques to enhance BPs that exhibit goal-oriented behaviors." Communications of the ACM **40**(10): 99-101.

Minsky, M. L. (1988). The society of mind. New York, Simon & Schuster.

Nielson, H. (2000). Introduction to SOAP. OOPSLA XML Process and Objects Symposium. 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis, ACM.

Patel, D., J. V. Sutherland, et al. (1998). Business Object Design and Implementation II: OOPSLA'96, OOPSLA'97 and OOPSLA'98 Workshop Proceedings. London, Springer-Verlag.

Paul, S., E. Park, et al. (1997). RainMan: A Workflow System for the Internet. IBM T.J. Watson Research Center.

Paul, S., E. Park, et al. (1998). Essential Requirements for a Workflow Standard. Business Object Design and Implementation II: OOPSLA'96, OOPSLA'97, and OOPSLA'98 Proceedings. D. Patel, J. Sutherland and J. Miller. London, Springer-Verlag**:** 100-108.

Schmidt, M.-T. (1998). Building Workflow Business Objects. Business Objects Design and Implementation II: OOPSLA'96, OOPSLA'97 and OOPSLA'98 Workshop Proceedings. D. Patel, J. Sutherland and J. Miller. London, Springer-Verlag.

Sutherland, J. (1998). Tutorial: Objects, Databases, and the World Wide Web. COMDEX/Object World, San Francisco, COMDEX.

Sutherland, J. and S. Alpert (1999). "Big Workflow" for Enterprise Applications. OOPSLA Workshop on the Implementation and Application of Object-Oriented Workflow Management Systems II., Denver, CO.

Sutherland, J. and J. McKenna (1993). Ensembles. Easel Corporation, Burlington, MA.

Van den Enden, S., E. Van Hoeymissen, et al. (2000). A Case Study in Application Integration. OOPSLA Business Object and Component Workshop. 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis.

Wegner, P. (1995). "Interactive Foundations of Object-Based Programming." IEEE Computer **28**(10): 70-72.