Knowledge Management Enterprise Services (KMES)
Concepts and Implementation Principles

Jens Pohl, Ph.D.

Executive Director, Collaborative Agent Design Research Center (CADRC)
California Polytechnic State University (Cal Poly)
San Luis Obispo, California, USA

Abstract

The purpose of this paper is to present concepts and implementation principles related to the design and development of reusable software services that are capable of assisting users at the operational level. Knowledge Management Enterprise Services (KMES) are an implementation of the service-oriented architecture paradigm, with a focus on the exchange of data within the meaningful context of a particular application (i.e., knowledge) domain. This requires a KMES service to incorporate a high level representation of this knowledge domain in the form of an ontology that is shared among all collaborating services within the application environment and at the same time specialized to the perspective that is appropriate to the servicing capabilities of the particular KMES service.

Although KMES services can operate in any distributed system environment, they represent a step toward semantic web services by incorporating many of the same objectives, such as self-sufficiency, interoperability, discovery, asynchronous interaction with clients, and context-based intelligence. Therefore, this paper also deals briefly in an Appendix with the notion of *web enabled* and the different types of thin-client user-interfaces that are prevalent today.

Finally, the paper discusses the software development process of a software system environment that maximizes the use of KMES services. Based on our CADRC Center's experience with the development of mostly military decision-support systems incorporating collaborative software agents, such KMES-based systems offer several advantages including a significant reduction in development time, decreased software development costs, and higher quality end-products.

Keywords

agents, context, data, data-centric, decision-support, design, development process, evaluation, information, information-centric, intelligence, KMES, knowledge management enterprise services, ontology, representation, semantic web, service-oriented architecture, software, thin-client, web enabled, web services.

The Service-Oriented Architecture Paradigm

The notion of *service-oriented* is ubiquitous. Everywhere we see countless examples of tasks being performed by a combination of services, which are able to interoperate in a manner that results in the achievement of a desired objective. Typically, each of these services is not only *reusable* but also sufficiently *decoupled* from the final objective to be useful for the performance of several somewhat similar tasks that may lead to quite different results. For example, a

common knife can be used in the kitchen for preparing vegetables, or for peeling an orange, or for physical combat, or as a makeshift screwdriver. In each case the service provided by the knife is only one of the services that are required to complete the task. Clearly, the ability to design and implement a complex process through the application of many specialized services in a particular sequence has been responsible for most of mankind's achievements in the physical world. The key to the success of this approach is the *interface*, which allows each service to be utilized in a manner that ensures that the end-product of one service becomes the starting point of another service. In this way the adapter that is required when you take your laptop computer on a business trip to another country interfaces between the foreign electrical outlet and the electric plug of your computer, which in turn interfaces with the electric cables that interface with the power unit, and so on.

In the software domain these same concepts have gradually led to the adoption of Service-Oriented Architecture (SOA) principles. While SOA is by no means a new concept in the software industry it was not until web services came along that these concepts could be readily implemented (Erl 2005). Initial attempts to provide the required communication infrastructure, such as the Distributed Computing Environment (DCE) and the Common Object Request Broker Architecture (CORBA) did not gain the necessary general acceptance (Mowbray and Zahavi 1995, Rosenberry et al. 1992). Web services and SOA are similar in that they both support the notion of discovery (Gollery 2002). Web services employ the Universal Description Discovery and Integration (UDDI) mechanism for providing access to a directory of web services, while SOA services are published in the form of an Extensible Markup Language (XML) interface.

So what is SOA? In the broadest sense SOA is a software framework for computational resources to provide services to customers, such as other services or users. The Organization for the Advancement of Structured Information (OASIS)[1] defines SOA as a "… *paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains"* and *"…provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects with measurable preconditions and expectations"*. This definition underscores the fundamental intent that is embodied in the SOA paradigm, namely *flexibility*. To be as flexible as possible a SOA environment is highly modular, platform independent, compliant with standards, and incorporates mechanisms for identifying, categorizing, provisioning, delivering, and monitoring services.

The Existing Web Services Environment

Web services are a particular implementation of the SOA paradigm. According to the World Wide Web Consortium (W3C) a web service may be defined as "… *a software application identified by a Uniform Resource Identifier (URI), whose interfaces and bindings are capable of being defined, described, and discovered by XML artifacts*"[2]. Currently most web services interact with other services or users utilizing the Hyper-Text Transfer Protocol (HTTP) to exchange XML-based messages defined in the Service Oriented Architecture Protocol (SOAP).

[1] OASIS is an international organization that produces standards. It was formed in 1993 under the name of SGML Open and changed its name to OASIS in 1998 in response to the changing focus from SGML (Standard Generalized Markup Language) to XML (Extensible Markup Language) related standards.

[2] See web site at: http://www.w3.org/TR/wsa-reqs#IDAIO2IB.

The SOAP standard defines an XML language and a set of rules for formatting objects and data that are independent of the programming language, operating system, and hardware platform.

Existing web service environments such as Microsoft's '.Net' software (Thai 2003, Chappell 2006) typically comprise a web server that utilizes HTTP for communication, UDDI as part of the standard definition of web service registries, a registry that already contains an entry for the accessing application, and any number of web services designed to facilitate some of the operations that the accessing application may wish to perform. In this respect, current web service environments rely on the notion of *predefined registrations and discovery* and do not support the notion of *opportunistic discovery*. UDDI, an international standard that defines a set of methods for accessing a registry, is structured to provide information about organizations, such as: who (about the particular organization); what (what services are available); and, where (where are these services available). However, UDDI does not provide descriptions of the available services in a semantic form that can be automatically interpreted by software (e.g., software agents), rather the descriptions are hard-coded or subject to human interpretation.

Communication between an application and a web server is almost always initiated by the application (i.e., the application sends a request and the web server sends a response). Specifically, the Uniform Resource Locator (URL) contains an identification of the particular web server to be used. This web server then finds the HTML page that corresponds to the URL and returns that page in the response. Immediately after the response has been sent the connection between the application and the web server is terminated and only reactivated if another response is requested. In this way a web server is able to handle many concurrent requests from applications.

Adding Meaning to Web Services

There are several reasons why computer software, and therefore web services, must increasingly incorporate more and more *intelligent* capabilities (Pohl 2005). Perhaps the most compelling of these reasons relates to the current data-processing bottleneck. Advancements in computer technology over the past several decades have made it possible to store vast amounts of data in electronic form. Based on past manual information handling practices and implicit acceptance of the principle that the interpretation of data into information and knowledge is the responsibility of the human operators of the computer-based data storage devices, emphasis was placed on storage efficiency rather than processing effectiveness. Typically, data file and database management methodologies focused on the storage, retrieval and manipulation of data transactions, rather than the *context* within which the collected data would later become useful in planning, monitoring, assessment, and decision-making tasks.

What are the enabling facilities that will allow software to interpret the meaning of data as an intelligent partner to the human user? This is a question that has engaged our CADRC Center in intensive explorations for the past 20 years. Several years before the advent of the Internet and the widespread promulgation of SOA concepts we started building distributed software systems of loosely coupled modules that were able to collaborate by subscription to a shared information model. Today, our KMES components are based on the same foundational principles to enable them to function as decoupled services. These principles include:

- An internal *information* model that provides a usable representation of one or more of the following three information areas: (a) the application domain in which the service is being offered; (b) the internal operational domain of the software application itself; and, (c) the role of the service within the external environment. In other words, the context provided by the internal information model must be adequate for the software application to perform as a useful adaptive set of tools in its area of expertise, be able to monitor and diagnose its own internal operational state, and describe its nature in response to external inquiries.
- The ability to *reason* about events within the context provided by the internal information model. Eventually these reasoning capabilities should extend beyond the ability to render application domain related services and perform self-monitoring maintenance and related operational efficiency tasks, to the ability of a service to be able to describe its capabilities and understandings to other external parties (i.e., other services and human users).
- Facilities that allow the service to *discover* other services and understand the nature and capabilities of these external resources.
- The ability of a service to *learn* through the acquisition and merging of information fragments obtained from external sources with its own internal information model. In other words, the internal information model must be dynamically *extensible*.
- The ability of a service to understand the notion of *intent* (i.e., goals and objectives) and undertake self-activated tasks to satisfy its intent. A typical relatively simple intent might be the objective of finding another service application capable of providing a specific service such as a weather forecast or an available weapon for destroying a given target. Far more sophisticated would be an objective that is only vaguely defined, such as the solution of a problem for which the solution approach is known in general terms only (e.g., locate the likely position of an enemy unit that has not been sighted for 24 hours).
- The ability of a service to increase its capabilities by either *generating* new tools (e.g., creating new agents or cloning existing agents) or *searching* for external assistance.

The Concept of Knowledge Management Enterprise Services

Knowledge Management Enterprise Services (KMES) are *self-contained* software components that offer their capabilities as services to external service requestors. Whereas in a SOA-based software system the available services normally operate at a lower system level as enablers of higher level functional capabilities, a KMES component is the incarnation of one or more of those functional capabilities. In other words, KMES components are services that operate at the functional level of the application domain.

They are designed to be platform independent and adaptable to a variety of applications. It is this adaptability that promotes their high degree of *reusability*. Some KMES components may have quite narrow capabilities such as the reformatting of weather data into a software interpretable weather forecast, while others will incorporate larger functional domains such as the optimum routing of goods from multiple origins along alternative routes to multiple destinations.

However, all of the services that operate within a given application domain are closely aligned to the knowledge context of that domain by sharing the same information model. While the software code of a KMES component is reusable, its internal information model needs to be reconfigured as it is moved from one application domain to another. This ensures that the KMES services within any particular application environment are able to exchange data within the functional context of that environment.

For example, in the transportation domain the optimum routing of goods from multiple origins along alternative routes to multiple destinations would include the following KMES components, providing their services within the common context of a shared information model:

- Conveyance load-planning (i.e., ships, barges, trucks, railcars, and aircraft of various types).
- Packaging of different kinds of shipping units (e.g., containers, pallets).
- Storage management in marshalling yards and warehouses.
- Route planning and re-planning.
- Map-based presentation for geospatial tracking.
- Scheduling.
- Interoperability bridges to external data feeds and other applications.
- Graphical and textual report generators.

This is in stark contrast to the large software systems that have been developed in the past and that invariably lead to a stove-piped architecture with almost insurmountable interoperability problems. Typically, in the case of these legacy systems the above functional capabilities have required the development of several systems with considerable duplication (e.g., user-interfaces, persistence facilities, and report generation) and largely incompatible data schemas.

A KMES-based system governed by SOA principles, on the other hand, is intended to meet several technical objectives that are aimed at maximizing horizontal and vertical interoperability. First and foremost, a KMES is designed to be as self-sufficient as the state of current technology will allow. Ideally, self-sufficiency should include platform independence with self-installing, self-configuring, and self-scaling capabilities. Second, it must incorporate discovery capabilities. However, discovery capabilities that are truly useful will require some degree of built-in intelligence. The combination of self-sufficiency, discovery and intelligence is potentially very powerful since it supports interoperability at the information level. In other words, KMES components are able to exchange data in the *context* that is provided by the shared virtual model of a particular real world knowledge domain.

Third, a KMES incorporates intelligent tools in the form of agents that support meaningful human-to-agent and agent-to-agent collaboration. These agents rely on the context provided by the internal information model to an even greater degree than do the discovery capabilities. In both cases the availability of context is a prerequisite for automated reasoning capabilities that can be applied to the interpretation of data changes, the opportunistic analysis of events, the spontaneous search for additional resources, and the generation of warnings and alerts.

Fourth, a KMES is capable of exposing functionality through objectified, domain-centric client interfaces. To take full advantage of this capability the KMES must support asynchronous

interaction with its external clients. This in turn requires adherence to industry-standard patterns such as JavaBeans[3], Property Change Management[4], and so on.

Fifth, by virtue of its internal information model a KMES is readily adaptable to operate in terms of application-specific notions and concerns. However, this also means that when reference is made to the *reusability* of a KMES then this refers to the software code and not the internal information model. Reconfiguring a KMES to a new application domain will involve initialization with the ontology of the new knowledge domain. Capitalizing on their decoupled nature, KMES components can be replaced with improved versions as the technology advances.

In summary, KMES modules are adaptable, self-contained software components that are capable of performing specific tasks within a net-centric environment. Service-oriented KMES capabilities typically take the form of distributable services whose functionality is exposed to potential clients as domain-centric objects employing key industry-standards. In other words, interaction between KMES services and their clientele occurs in terms of object-level operations (i.e., object creation, property modification, etc.) over the domain model exposed by such services. Such object-level manipulation is partnered with the asynchronous notification of events to interested parties (i.e., clients as well as the KMES service itself). For example, consider a KMES Route Planning service. Such a service could be invoked by a client creating a set of domain objects (e.g., *requirements*, *constraints*, and *route topology* objects) defining the context of their request. Listening to such objects, the KMES Route Planning service responds by processing this context into a solution. This solution would, in turn, be exposed as a set of objects based on the service's domain model interface. In the same asynchronous manner by which the service became aware of the client's initial request, the client in turn will receive its results through object-level event notification.

KMES as a Net-Centric Architecture

The expressive, context-rich representation upon which many KMES capabilities are built together with the significant potential for higher levels of decision-support lends itself to incorporation of intelligent agent technology. When equipped with such enabling features, agents can collaborate with users to assist in formulating solution alternatives, compare and contract their associated costs, and aid in successful execution through constant monitoring and the performance of necessary mediation. For example, agents in a military logistical domain can receive status reports, track shipments, incorporate suitable and available assets in plans, and provide appropriate updates on location and security risks. Others may track the path of incidence and provide appropriate graphic and textual updates for action. Finally, agents can interpret incoming signals, identify significant events (i.e., changes), and modify proposals to meet the changing situation as it develops. The vision of such *intelligent agents* is quite

[3] JavaBeans are reusable software components (i.e., classes) written in the Java computer language. Based on certain conventions (i.e., naming, construction and behavior) a JavaBean is used to encapsulate several objects into a single object that can then be passed around.

[4] Property Change Management refers to the ability to access diverse documents across a network (i.e., Internet or intranet) without the need for manual intervention. For example, the Web Interface Definition Language (WIDL) automates interactions with HTML/XML documents and thereby allows the Web to serve as an integration platform.

compelling and it is now generally believed to be a critical component for successfully harnessing the increasing complexities of a *net-centric* environment.

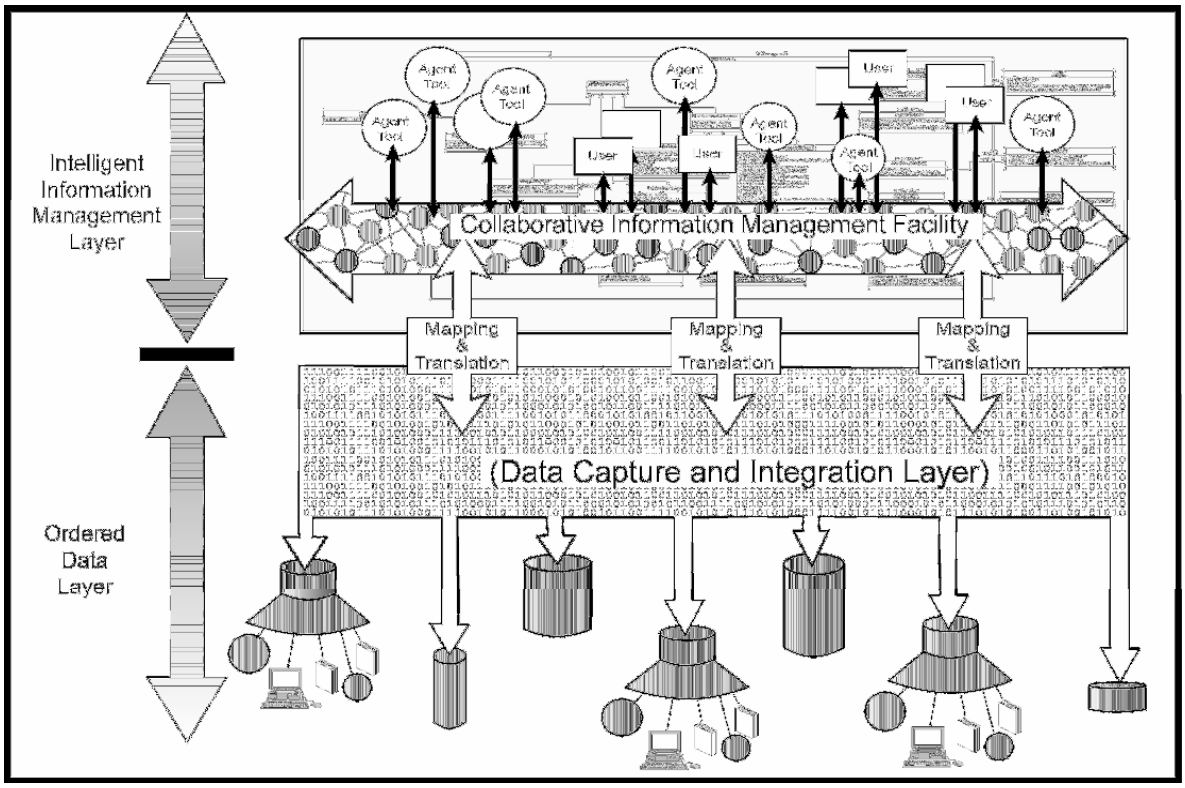

Figure 1: Conceptual KMES-based net-centric architecture

Existing data-centric systems lacking the adaptive, interoperability characteristics described above can be integrated into such an agent-empowered KMES software environment through the use of *interoperability bridge facilities* that effectively map the data model in one system to the information model of the other. This allows for a meaningful bi-directional interaction and exchange of data in context. Such bridges have been successfully demonstrated by military organizations for linking legacy data-centric systems to intelligent command and control systems (Pohl et al. 2001). The technology is inherently scalable and allows for the efficient and effective interconnection of multiple participants within a heterogeneous net-centric environment.

Conceptually, an intelligent net-centric software environment typically requires the seamless integration of a KMES-based information management facility with existing data sources. This can be achieved with an *information-centric* architecture that consists essentially of two components (Figure 1): a data-centric Data Capture and Integration Layer that incorporates linkages to existing data sources; and, an Intelligent Information Management Layer that overlays the data layer and utilizes software agents with automatic reasoning capabilities, serving as decision-support tools.

The Intelligent Information Management Layer architecture (Figure 1) utilizes intelligent software agents capable of collaborating with each other and human operators in planning, re-planning, monitoring, and associated decision-support environments. Typically such intelligent systems are based on software development frameworks, such as the ICDM (Integrated

Cooperative Decision Making) and TIRAC™ (Toolkit for Information Representation and Agent Collaboration) software development frameworks used by CDM Technologies and the Collaborative Agent Design Research Center (CADRC) at Cal Poly[5] for the development of military and commercial systems, respectively (Pohl et al. 2004a and 2004b).

Data Capture and Integration Layer: The bottom layer of the system takes the form of an operational data store and/or Data Warehouse, implemented within a commercial off-the-shelf relational database management system (RDBMS). This repository integrates data extracted on a periodic basis from several external sources into a *common* data schema. Although not a requirement, the design of the data schema is typically closely modeled on the structure of the ontology of the Intelligent Information Management Layer to minimize the required data-to-information and information-to-data mappings between these two system layers. Further, to facilitate an object-oriented environment, content managed by the Data Capture and Integration Layer is exposed to its information-oriented clients (e.g., KMES-based environments) as objects rather than relational tables. Translation between these two forms is typically accomplished through employment of some form of Object Relational Mapping (ORM) technology.

In conformity with normal enterprise data management practices the Data Capture and Integration Layer incorporates the following four characteristics:

- It is subject-oriented to the specific business processes and data domains relevant to the application area (e.g., goods movement across national borders or tactical command and control in a military theater).
- It is integrated so that it can relate data from multiple domains as it serves the data needs of the analysis functions performed by collaborative agents in the Intelligent Information Management Layer.
- It is periodically synchronized with events and changes occurring in the external data sources from which it derives its content.
- It is time-based to support the performance of analyses over time, for the discovery of patterns and trends.

A multi-tier architecture is used to logically separate the necessary components of the data layer into levels. The first tier is the RDBMS, which ensures the persistence of the data level and provides the necessary search, persistence, and transaction management capabilities. The second tier is the service level, which provides the interface to the objectified data level and at the same time supports the data access requests that pass through the mapping interface from the Intelligent Information Management Layer to the Data Capture and Integration Layer. It is designed to support request, response, subscribe, and publish functionality. The third tier is the control level, which routes information layer and user requests to the service level for the update, storage and retrieval of data. Finally, a view layer representing the fourth tier serves as a graphical user-interface for the Data Capture and Integration Layer.

Information Management Layer: The Intelligent Information Management Layer consists of KMES components in the form of a group of loosely coupled and seamlessly integrated decision-support tools. The core element of each KMES component is typically an ontology that

[5] ICDM and TIRAC™ are software development toolkits developed by principals of the CADRC Center at California Polytechnic State University, San Luis Obispo and its commercial arm CDM Technologies, Inc.

provides a relationship-rich and expressive model of the particular domain over which the KMES capability operates. Normally, KMES components are based on a three-tiered architecture incorporating technologies, such as distributed-object servers and inference engines, to provide a framework for collaborative, agent-based decision-support that offers developmental efficiency and architectural extensibility. The multi-tiered architecture clearly distinguishes between information, logic, and presentation. Most commonly an *information tier* consists of a collection of information management servers (i.e., information server, subscription server, etc.) providing domain-oriented access to objectified context, while a *logic tier* houses communities of intelligent agents, and a *presentation tier* is responsible for providing meaningful interfaces to human operators and external systems.

A Typical KMES Ontology

This section discusses a portion of a domain-centric ontology upon which a particular logistics-oriented KMES capability may operate. It should be made clear, however, that while in this case the ontology deals with logistic operations, the domain, scope and expressiveness within that domain, as well as the bias (i.e., perspective) of the model is driven by the use-cases that are supported by the KMES capability, as well as the subject matter those use-cases operate over. In the example provided below, the underlying KMES ontology is divided into several somewhat related domains (Figure 2). While some of these domains describe application-specific events and information (e.g., goods movement transactions, shipping routes, and so on) others describe more general, abstract notions (e.g., event, threat, view, privacy). The goal in developing such an ontology is to abstract general, cross-domain notions into high-level, extensible domain models. As such, these descriptions can be refined and specialized across several application sub-domains. In other words, more domain-specific, concrete notions can be described as extensions of these abstract models.

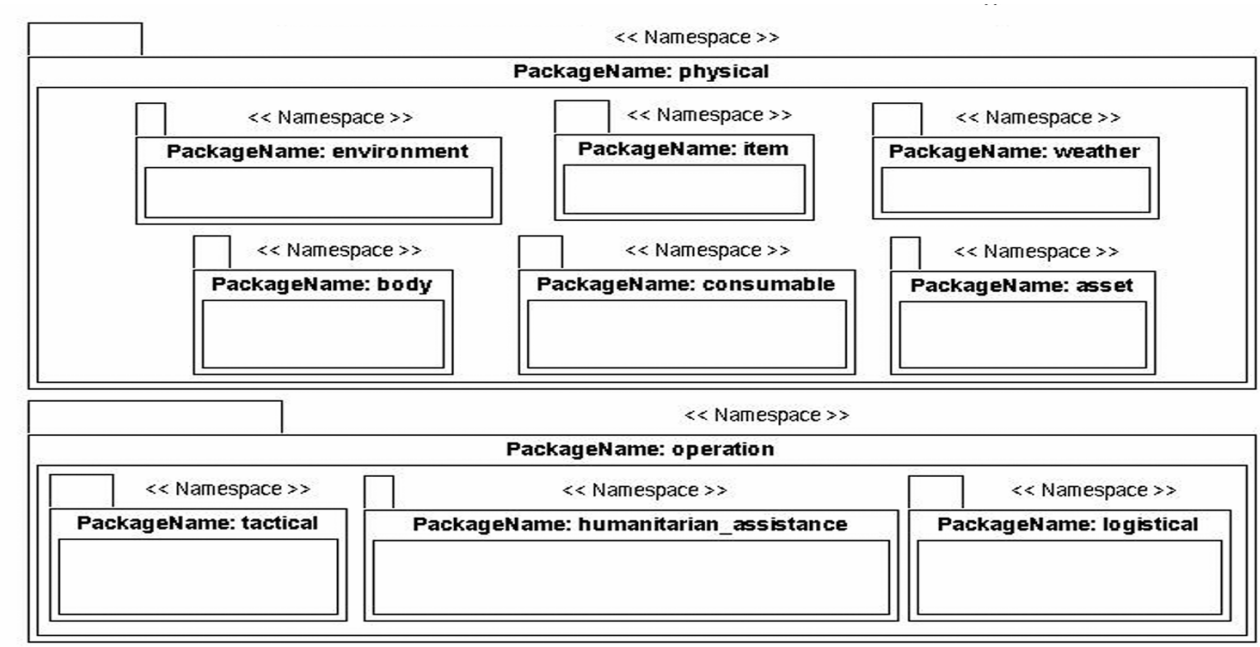

Figure 2: Typical ontology domains within a military application area.

Accordingly, a KMES ontology includes several primary meta-characteristics. Through mechanisms such as inheritance as well as the application of underpinning analysis patterns, these meta-characteristics can be propagated to more specific ontological components. To illustrate, the simple application of inheritance allows, for example the abstract characteristic of something being trackable to be propagated into more specialized entities. Applied to this logistics example, if such a trackability notion is introduced at the *physical.Mobile* level then, through inheritance, any entity that is a kind of *physical.Mobile* automatically receives the property of being *trackable*. Taking this example further, a second meta-characteristic may relate to the dispensability of an item. If this property is represented at the *physical.Item* level then, similar to the *trackable* characteristic, anything that is a kind of a *physical.Item* automatically receives the quality of being dispersible or *suppliable*. In addition, as an extension of *physical.Mobile*, such *suppliable* items are also *trackable*. It can be readily seen that together these two meta-characteristics provide an effective foundation for propagating fundamental notions to hierarchically related definitions. Although inheritance can be a useful mechanism for the propagation of fundamental characteristics to more specific classifications, an even more powerful, and oftentimes less restrictive, technique is the application of extensible analysis patterns. Such patterns offer adaptable model fragments, thereby providing a fundamental definition of the notion being represented in the form of an extensible model architecture for applying this underpinning concept to other elements of a domain model. Such patterns typically employ a *role* metaphor, where elements of a model may essentially *play the role* of something embodying the fundamental notion or characteristic.

The Capabilities of KMES Agents

KMES components equipped with intelligent agents may employ a variety of framework technologies and reasoning paradigms to execute their agent-based logic. Regardless of the specific agent technology employed, their capabilities can exist at a monitoring, largely reactive level, or at a higher consequential and proactive level. In actuality, the event-oriented nature of the former may, in fact, trigger the proactive reasoning of the latter. In the context of homeland security, for example, such reasoning may produce: a warning[6] that hazardous material is en route; a warning that a truck has not reached a waypoint within a certain time limit; an alert that a truck has not reached a waypoint within a more critical time limit; a warning that a truck is near a higher risk area; an alert that a truck has stopped for more than a certain time near a higher risk area; an alert that the loaded weight of a truck does not match the final weight at the border check point; and so on.

Within the same homeland security context (i.e., specifically inland border control), typical higher level agent inferencing capabilities may include warnings and alerts that a particular combination of circumstances involving encyclopedic data and truck-based or convoy-based confirmation data entered at waypoints and checkpoints constitutes a higher risk situation. Examples include, a particular driver transporting certain kinds of goods, or the combination of an authorized substitute driver taking an authorized alternative route without apparent reason, and taking a significantly longer time between two consecutive checkpoints. While none of these

[6] Typically, agents will communicate with the user at different levels of urgency. For example, a warning may simply draw the user's attention to some particular event or situation, while an alert signifies that the user's focused attention is urgently required.

individual anomalies might be sufficient to cause concern, their combined occurrence may well constitute a risk requiring further actions.

Comparing the Development of Legacy and KMES-Based Systems

Considerable time and cost savings can be realized in the KMES approach, without sacrificing quality. In fact, the quality of the software developed can increase due to both the extensive formal validation and verification process appropriate for a core capability as well as the informal validation and verification resulting from its repeated use in the field. This is readily seen when we compare the development life-cycle of a legacy software system (Figure 3) with a KMES-based system (Figure 4).

Software development projects, whether legacy or KMES-based, commence with the recognition of a need. Typically this is a functional need that has been identified by an operational failure or through some form of analysis driven by a desire to achieve a higher level of effectiveness in supporting certain operations. This is followed by the formulation of an end-state vision and, if this vision in conjunction with the need are sufficiently compelling, a decision to act. Once that decision has been made the translation of the end-state vision into a set of use-cases on the basis of which the actual product requirements are formulated. While both the initial level of detail contained in the use-cases, consequential requirements specification, and the degree of involvement of the development team in the formulation of these two artifacts will vary with the type of project and the kind of development process adopted (e.g., prescriptive, agile), there is an undisputed need for some form of formalized documentation describing these various aspects.

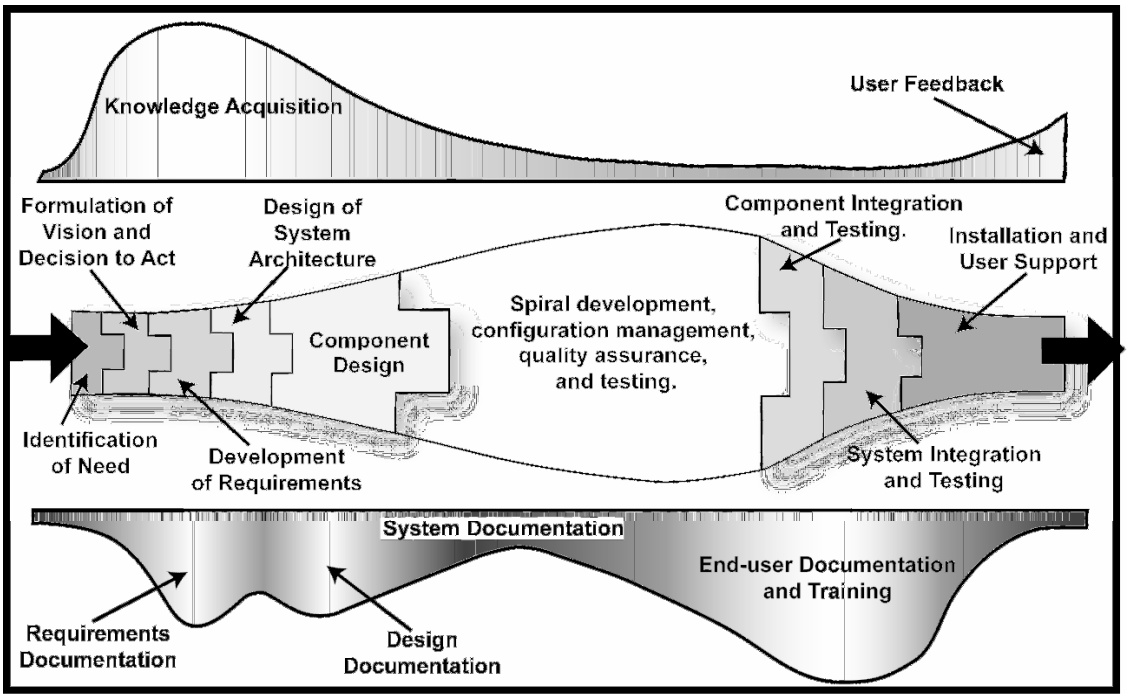

Figure 3: Development life-cycle of legacy software

Up to this point, as shown in Figures 3 and 4, there appears to be little difference between the legacy and the KMES-based software development approaches. In either case the degree to

which the software marketing and development teams will be permitted to assist the customer and end-user in establishing end-state objectives and requirements depends largely on factors that are only indirectly related to the development approach. One of these factors is that as a result of rapid advances in information technology the potential users of this technology are often not aware of the kind of progressive capabilities that could significantly improve the efficiency and effectiveness of their efforts.

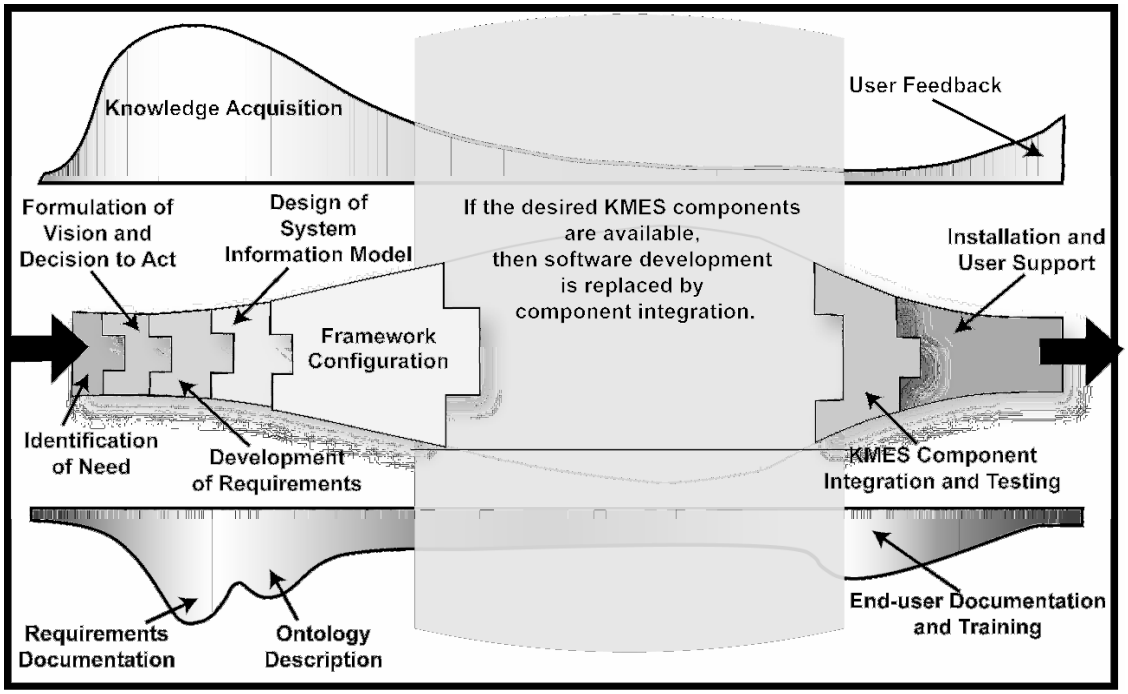

Figure 4: Development life-cycle of KMES-based software

Once the requirements have been established the development process of legacy software will normally proceed with the design of the system architecture (Figure 3). However, in the case of a KMES-based solution the mature design and engineering principles have already produced an open architecture exploiting the semantic-rich representation (i.e., *context*) and well-structured interface protocols that will allow the KMES components to effectively interact (Figure 4). Therefore, the substitute step in the KMES-based approach is the design of aspects specific to the application logic, representation, and presentation, each of which may capitalize on supportive building blocks offered by the KMES capabilities being employed. In fact, much of this effort will essentially take the form of *adapting* off-the-shelf KMES capabilities to operate in terms of application-specifics.

In the case of legacy software formulation of application-specific logic, representation, and presentation is in addition to the formulation of the core functionality that would otherwise have been taken care of by inclusion of KMES components. Further, the legacy design and implementation of such core functionality is typically tailored to the specific application leaving little opportunity for reuse. As a result, if the required KMES components are available then the normal time consuming and costly development cycle of traditional software is avoided and replaced by the relatively simple process of integrating KMES components into the target operating environment. Even with the adoption of a spiral development cycle, the traditional software development cycle can take years and account for as much as 60% of the total software

production cost and time. In the case of KMES-based software the component integration stage can be completed in a matter of a few months and sometimes weeks. As a result, the delivery of the first set of usable capabilities can be reduced to six months or less after the initiation of the software development project.

KMES Benefits to the Customer

The principal benefits of KMES-based software systems are threefold: early delivery of usable decision-support tools; decreased software acquisition costs; and, higher quality products. In the experience of our Center, which is predominantly engaged in the design and development of intelligent software systems for national security applications, the considerable time savings that can be achieved with the KMES approach has been of particular interest to our military customers. This is probably due to their increasing focus on *adaptive planning* capabilities. In this military context *adaptive planning* is defined by the Adaptive Planning Roadmap[7] as the capability to create and revise plans rapidly and systematically, as circumstances require.

Our military customers quickly realized that the adaptive planning mandate will require new planning and decision-support tools with superior capabilities (i.e., intelligence) that can be rapidly implemented, and are extensible and replaceable to accommodate the evolving needs of the user community. KMES-based software systems have the potential for meeting this challenge by virtue of the following inherent advantages:

- *Rapid delivery* of meaningful capabilities, with the potential of achieving a first usable product installation within three to six months after the initiation of a software development project.
- *Lower cost* due to the replacement of the normally prolonged software development period with a much shorter KMES integration period.
- *Greater reliability* and quality due to exhaustive core-component verification and validation in conjunction with the maturity that comes from extensive in-field use.
- *Interoperability* through component design based on standard protocols and a decoupled, multi-tiered framework.
- *Flexibility* to extend functional capabilities through plug-and-play components and an open architecture.
- *Multiple deployment options* including net-centric delivery alternatives of hosted-managed services.

[7] Adaptive Planning Implementation Team (ODASD/JOWPD); 'Adaptive Planning Roadmap'; Version 1, Final Draft, 3 January 2005 and 'Adaptive Planning and Execution Roadmap III', Draft 8 February 2007, Joint Chiefs of Staff, US Department of Defense.

References

Chappell D. (2006); 'Understanding .NET'; 2nd Edition, Independent Technology Guides, Addison-Wesley, Boston, Massachusetts.

Erl T. (2005); 'Service-Oriented Architecture (SOA): Concepts, Technology, and Design'; Prentice Hall Service-Oriented Computing Series, Prentice Hall, Englewood Cliffs, New Jersey.

Gollery S. (2002); 'The Role of Discovery in Context-Building Decision-Support Systems'; Office of Naval Research Workshop on Collaborative Decision-Support Systems, Quantico, Virginia, 18-19 September (Proceedings available from CADRC Center, Cal Poly, One Grand Avenue (Bdg. 117T), San Luis Obispo, California 93407).

Mowbray T. and R. Zahavi (1995); 'The Essential CORBA: Systems Integration Using Distributed Objects'; Wiley, New York, New York.

Pohl J., M. Porczak, K.J. Pohl, R. Leighton, H. Assal, A. Davis, L. Vempati and A. Wood, and T. McVittie, and K. Houshmand (2001); 'IMMACCS: A Multi-Agent Decision-Support System'; Technical Report, CADRU-14-01, Collaborative Agent Design (CAD) Research Center, Cal Poly, San Luis Obispo, CA, June. (2nd Edition)

Pohl J., K. Pohl, R. Leighton, M. Zang, S. Gollery and M. Porczak (2004a); 'The ICDM Development Toolkit: Purpose and Overview'; Technical Report CDM-16-04, CDM Technologies, Inc., San Luis Obispo, California, USA (May).

Pohl J., K. Pohl, R. Leighton, M. Zang, S. Gollery and M. Porczak (2004b); 'The TIRAC™ Development Toolkit: Purpose and Overview'; Technical Report CDM-17-04, CDM Technologies, Inc., San Luis Obispo, California, USA (August).

Pohl J. (2005); 'Intelligent Software Systems in Historical Context'; in Jain L. and G. Wren (eds.); 'Decision Support Systems in Agent-Based Intelligent Environments'; Knowledge-Based Intelligent Engineering Systems Series, Advanced Knowledge International (AKI), Sydney, Australia.

Rosenberry W., D. Kenney and G Fisher (1992); 'Understanding DCE'; O'Reilly and Associates, Sebastopol, California.

Thai T. L. (2003); '.NET Framework Essentials'; 3rd Edition, O'Reilly, Sebastopol, California.

Appendix: The Multiple Meanings of *Web Enabled*

The term *web enabled* is widely used with somewhat differing intended meanings. The following explanation of this term was prepared as an internal communication by Steve Gollery, a Senior Software Engineer in our CADRC Center. It is reproduced in abbreviated form in this paper as an appendix for clarification purposes.

> The phrase *web enabled* is sufficiently vague that it provides little guidance in understanding the intent of the persons and organizations using it. This lack of definition becomes critical in view of the fact that some meanings of *web enabled* severely limit the capabilities of client-side software. The following list of the possible meanings of *web enabled* may not be inclusive, but it does cover the main variations.

1. A *rich client*, communicating with its server(s) using HTTP, SOAP, and so on. In this configuration the capabilities of the rich client are essentially unconstrained since it is entirely resident on the user's computer. Performance considerations will largely dictate the pattern of interaction between the client and the server.
2. A client implemented as a *browser plug-in*. This also can be a fairly rich client, with many of the same kinds of interactions. It is likely that the use of a browser plug-in will allow the implementation of most if not all of the functionality of a rich client executing outside of a browser.
3. A client implemented as one or more *applets* running in a set of web pages. Applets are designed to be as secure as possible, since they are downloaded from a web site to the user's computer. Applets, for example, cannot read or write files on the user's computer, so the user-specific state must be stored on the web site that the applet came from and downloaded by the applet. The necessity of sending all the executable code along with the information may mean that functionality will need to be limited due to the time it takes to download. Applets are seen less often now than they were in the 1990s. Many uses of applets have now been replaced by JavaScript, Flash, and/or Ajax client-side code (see below for descriptions of these techniques).
4. *AJAX* as an acronym for Asynchronous JavaScript and XML, is a recently-coined term for a set of technologies that have been in use for some time. It is similar if not identical to Dynamic HTML (DHTML). The core concept is that the user interface to the application is provided by JavaScript running within the browser and interacting with a web server by sending and receiving XML documents. AJAX eliminates the need to completely replace web pages in response to user input and allows user interaction that can be more like a desktop application than a browser-based application. Building AJAX applications requires the use of multiple development languages and skills. Development toolkits for AJAX-style applications, such as the BackBase[8] toolkit, have become available in recent years. AJAX is useful for building applications that require high levels of interactivity in an environment that permits JavaScript.
5. *Flash*[9] is one of the standard plug-ins that nearly every browser user has downloaded at some time or another. Flash has a bad reputation in some quarters: probably everyone has at one time or another run into web sites that feature pointless Flash animations that take too long to download and delay us from getting the information we are looking for. But Flash also provides a highly interactive user experience, and is being used to produce web applications with a complete user interface. Like AJAX, Flash applications use scripting to implement application behavior, and for communication with the web server.
6. A client implemented as a set of *HTML pages*. This is the most limiting form of *web enabled*. With no executable code on the client, all changes to the information presented to the user must be accomplished by regenerating the web

[8] BackBase, San Mateo, California.

[9] Flash is a Macromedia product that is now owned by Adobe Systems Inc., San Jose, California.

page and sending the new version. This can only be accomplished in response to action in the browser. For instance, the user clicks on a button, causing the contents to be sent to the server in a *form*. The server constructs or loads a web page and sends that page as the response. It is not possible for the web server to *push* a new version of the page autonomously unless some other software is installed on the user's computer.

Of these techniques, the only one guaranteed to be able to handle all types of user interaction is the *rich client*. However, this is not what is normally meant by *web enabled*, since the kind of communication that is used between client and server is invisible to the user. In other words, from the user's point view the *rich client* would not seem to have anything to do with the web.

The popular notion of *web enabled* assumes the use of a web browser of some sort. From a development point of view, the preferred approach to building clients that run in a browser would likely be AJAX. But in high-security environments the downloading of JavaScript to a user's browser may not be permitted. In fact, there is little if any risk in running JavaScript. The language does not permit destructive behavior and security holes that existed in older browser versions have long been fixed. However, there is always the possibility that draconian security regimes may be in place in some prospective environments. In a worst case scenario, it would be necessary to fall back to using HTML only pages.

The HTML web page approach is really only suitable for viewing text and pre-defined images. The user would only be able to interact with the client and server by filling out *forms*, submitting them, and receiving the result. An HTML client would be able to view a graphic image such as a map in the kind of objectified geospatial framework that is commonly used by the military to track friendly and enemy forces. However, the graphic environment would not be interactive. For example, the user would not be able to drag and drop objects (e.g., infrastructure objects such as bridges, buildings, routes) from one location to another on the map. By contrast, in an AJAX or Flash-based application such interaction would require some effort on the part of developer, but it could be accomplished.

All forms of web-enabling have consequences for the possible functionality of a system, but some consequences are more severe than others. It is important to know what the end-users expect when they ask for a *web enabled* system, so that they can be made aware of the limitations of each of the alternative implementations.