Alternative Technologies

A

Technical Reference Architecture

for

Enterprise Application Integration

David McGoveran

Alternative Technologies
6221A Graham Hill Road, Suite 8001
Felton, California, 95018
Website: www.AlternativeTech.com
Email: mcgoveran@AlternativeTech.com
Telephone: 831/338-4621 Facsimile: 831/338-3113
Report Number 990315

Disclaimer and Notice

For information about this or other reports, or other products and services (including consulting and educational seminars), contact Alternative Technologies directly by telephone, mail, or via our Web site:

Alternative Technologies
13150 Highway 9, Suite 123
Boulder Creek, CA 95006
Telephone: 831/338-4621 FAX: 831/338-3113
Email: mcgoveran@AlternativeTech.com
Website: www AlternativeTech.com

---

1. Introduction

EAI (Enterprise Application Integration) is a complex effort that involves integrating many types of software (see Figure 1 on the next page). Most EAI projects are one-off, meaning that the problems are solved without planning for retaining either the knowledge gained, nor with an intention of being able to leverage the technologies developed, purchased, and used. The result is a high and repetitive cost across EAI projects.

Even when an EAI project is a part of a longer term effort to integrate software throughout a company, it can be extremely difficult to maintain a consistent view of the projects. From project to project, the software involved will differ significantly. In addition, the business processes and data that must be integrated will differ. Additionally, there are differences in staffing, which will almost always lead to multiple understandings of the problems and their solutions.

Another part of the reason for this is confusion is the different perspectives that EAI vendors and systems integrators have with respect to their products and services. Today, each EAI vendor has their own concept of EAI architecture that promotes the use of the greatest number of their products. Unfortunately, no EAI vendors has a complete solution. That is not a reflection on the breadth of the vendors offering, but rather a strong warning about the complexity of most corporate computing environments. Of course, when an integration project extends its goals beyond the corporate boundaries, the complexity increases combinatorially with each company included. Supply chain integration, customer relationship management, one-to-one marketing, and e-Commerce applications are examples of projects which have goals beyond corporate boundaries.

Although there are many problems that plague EAI projects, one that stands out as extremely important is the need for an EAI Technical Reference Architecture. The lack of a common architecture within which to identify, compare, and evaluate EAI approaches, technologies, and services prevents any coherent discussion, just as does the lack of a common language or a common integration methodology.

This report documents Alternative Technologies' efforts to provide an EAI Technical Reference Architecture. It is a work in progress and will be updated with new versions in the future. For obvious reasons, this report does not provide access to all of the available EAI Technical Reference Architecture information. Nonetheless, we have tried to supply

enough information to make this a useful document for those involved in EAI or other forms of application integration.

Software to be Integrated

- **packaged applications**
  - ***ERP***
  - ***CRM***
  - ***SFA***
- **proprietary applications**
- **legacy applications**
- **component applications**
- **Web-based applications**
- **middleware**
  - ***networks***
  - ***ORBs***
  - ***message brokers***
  - ***event brokers***
  - ***application servers***
  - ***Web servers***
  - ***TP monitors***
- **databases**
  - ***relational***
  - ***hierarchical***
  - ***network***
  - ***object***
  - ***hybrid***
  - ***flat-file***
  - ***proprietary***

Figure 1

---

2. Functional Block Architectures

Functional block architectures have a simple goal: to provide an organizing framework for the key functions of a system, in this case a software system. They do not describe functional requirements for a particular application or implementation. Rather, they are intended as conceptual guides when identifying the functional roles of candidate components for the design of a new system, or when analyzing an existing system for architectural consistency.

There are a number of rules that should be followed when developing a functional block architecture. In the remainder of this section, I will list those rules and their advantages.

- Use a hierarchy of diagrams to capture functional detail

The use of an initial high level of abstraction permits the framing of the architecture in simple, easily agreed upon terms. Subsequent elaboration of functionality is done in lower levels of detail, with each being a portion of the original

diagram (a "drill-down").

- Use rectangles, and only rectangles, to represent any type of functionality

Rectangles are reasonably simple, and can be fit together easily. That simplicity forces simplicity in the diagram.

- Limit the number of blocks within a diagram to a small number

This minimizes the complexity of each diagram, again helping with understandability and reaching consensus (or providing explanation) in a stepwise fashion.

- Label blocks descriptively, and define the label as a part of the documentation

The labels used in a functional block architecture are as important as diagrams. Each term or phrase used as a label should have a simple definition, used consistently throughout the architecture.

- Use only one block for any type of functionality

If two or more blocks have functionality in common, either combine the common functionality (thereby creating additional blocks with refined functionality) or combine these blocks, highlighting the common functionality in a higher level diagram. The use of redundant blocks in a diagram generally leads to implied assumptions regarding the interpretation of each block and how they are (physically) different. The diagram itself should make all such assumptions explicit. Examples of frequently implied differences are differences in physical location or differences due to contiguity such as "the DBMS used by accounting" vs. "the DBMS used by shipping".

- If any two blocks are nested in a single diagram, the outer box should not imply any distinct functionality

An outer block can be used to help clarify grouping of functionality, but the boundary does not represent a functional interface.

- Use contiguity (adjacency) of blocks to represent common interface requirements

If, for example, arrows are permitted for this purpose, the meaning of two blocks being contiguous is subject to multiple interpretations and the entire diagram can easily become impossible to understand. Likewise, this rule forces the implementation of interfaces to be simple and highly structured. This rule also forces the diagram to have a representation as a two-dimensional network in time and space. It forces all interfaces to be clearly identified. As a result, an estimate of the costs associated with any possible execution path between functional blocks is computable as the number of boundary crossings. (In a simple, non-distributed implementation of the architecture, each boundary crossing represents a function point or invocation.)

- Layers of design can be exploded, but there are no alternative explosions

With every alternative comes the possibility of confusion. When drilling down in an effort to examine detail, there should be a single hierarchy of diagrams.

- Use multiple rectangles at a lower level of the diagram hierarchy to refine functionality

This maintains the association between subtypes of a particular functionality and their more general type of functionality.

- Avoid the mention of physical resources, whether implicit or explicit

This permits either distributed or local implementation of functionality without prejudice. It also permits replication of services and functions.

- Use a consistent interpretation of the diagram across particular applications of the architecture

The permits multiple applications of the architecture to be subsequently viewed as a single application of it. Thus, two projects that use the same architecture can be guaranteed to support subsequent integration if the same facilities are used to support for key interfaces.

Checking Implementations

A functional block architecture is checked for consistency with a particular implementation by tracing the control flow through the system. Depending on the type of system, control can be passed, for example, by message, data, event detection, or invocation, or any combination of these. If a functional block architecture can correctly support (1) the functions, (2) interfaces, and (3) transfers of control present in an actual system, then it is a consistent functional representation of that system.

- Use single headed arrows between adjacent blocks to designate transfer of control, message passing, or component invocation in a particular implementation

An arrow should not pass through a boundary of a block more than once. It should originate within a block and terminate in a distinct second block.

- Label arrows with numbers to indicate any control flow sequencing

The numbers can be explained to identify what data, messages, or parameters are passed between blocks, what events trigger the transfer, and what business rules apply.

- Arrows should be only straight line segments between immediately adjacent block boundaries

Meandering arrows and arrows that travel through white space. This guarantees the simplicity of the architecture and especially the interfaces.

- Give each block a specific functional interpretation

The functionality of a particular block may be given a specific interpretation for a particular implementation of the architecture, including a functional "no-op", but must be consistent for that implementation.

- Use additional levels of refinement to implement a general or reference architecture

General or reference architectures are more useful the more broadly they may be applied. This may mean that certain functional details are best ignored, to be added for a particular implementation. However, such additions should be added only by refinement (i.e., additional level of diagram) in order that all interfaces remain intact and consistent with the reference architecture.

---

3. An EAI Technical Reference Architecture

Alternative Technologies' EAI Technical Reference Architecture is a functional block architecture. Version 1.0, presented here, is intended to provide a means for:

- organizing the architectures of EAI projects
- providing consistency across projects
- enabling subsequent integration among systems
- improving the retained value across EAI projects
- reducing the cost of EAI projects
- identifying missing EAI functionality

- evaluating the complexity cost of EAI implementations
- inter-relating multiple EAI vendors products
- identifying the functional contribution of an EAI vendors products

Alternative Technologies' EAI Technical Reference Architecture was created originally in recognition of the importance of EAI to our Zero Management Initiative. In particular, the Zero Management Technology requires an EAI Framework (see Figure 2). As can be seen, the EAI Framework divides integration into three levels or classes of activity, each supported by a type of modeling. In other words, integration becomes a design-oriented procedure rather than a one-off activity.

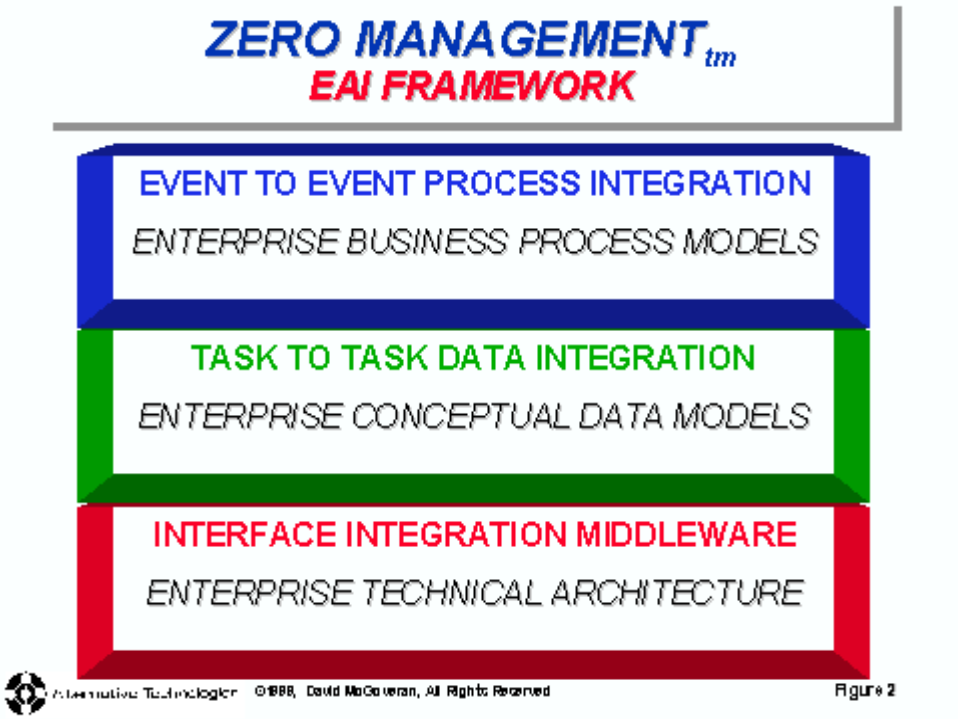

Figure 2

The top level of the Zero Management EAI Framework shows "Event to Event Process Integration" in recognition of the fact that business enterprises are driven by events, the permissible sequences of which form business process models. Thus, it is essential that business process models be integrated. The second level of the Zero Management EAI Framework shows "Task to Task Data Integration" in recognition of the fact that the response to an event is the execution of a task and that tasks are enabled by data. Thus, it is essential that conceptual or external data models (to use the traditional ANSI 3-schema terminology) be integrated. Of course, tasks are generally implemented by application services whether they be components or applications (packaged, custom, and legacy). The third and lowest level of the Zero Management EAI Framework addresses "Interface Integration Middleware" in recognition of the infrastructure required to support integrated processes and data. Such middleware obviously includes message brokers, TP monitors, ORBs, process engines, application servers, Web servers, network services, and the like. It also includes software commonly referred to as adapters or connectors (among other names). We prefer to call these pieces of software "Integration Agents", a term coined by Richard Skrinde and proposed for adoption by the EIC. As we shall see, this term is much more appropriate to the functionality actually required.

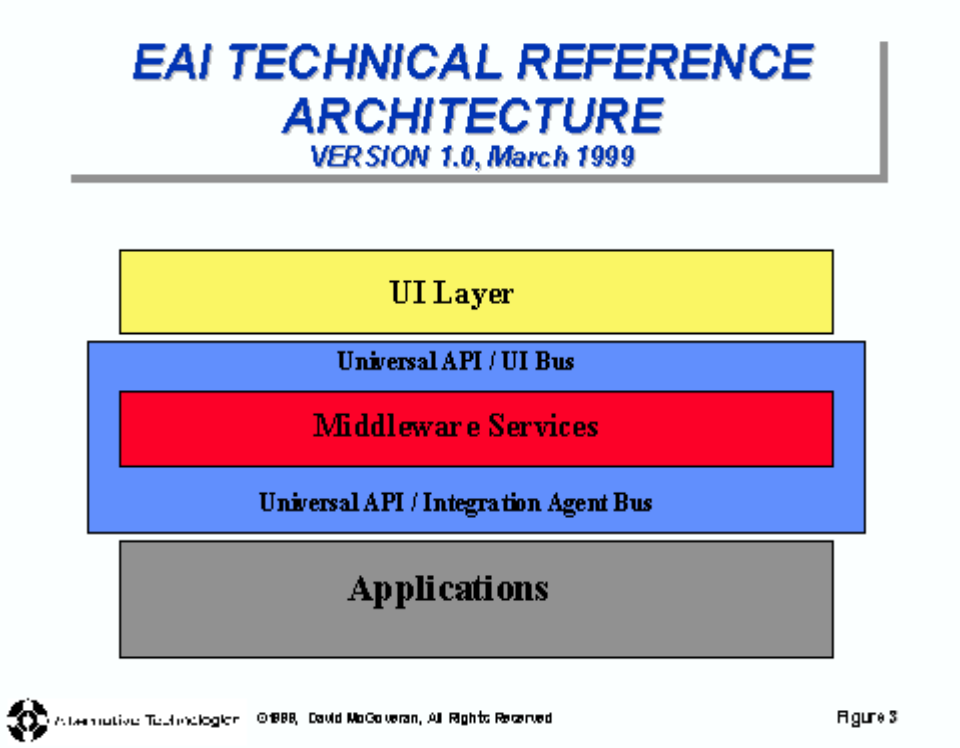

Figure 3

The EAI Technical Reference Architecture consists of four blocks at the highest level (see Figure 3). These are the UI (User Interface) Layer, a Universal API, Middleware Services, and Application Services. The UI Layer consists of all presentation functionality. The Universal API, here referred to as an API / Bus, has two portions. The first (at the top of the block) that provide interface adaptation functionality between Middleware Services and the UI Layer. The second (at the bottom of the block) provides interface adaptation functionality between Middleware Services and the Application Services. The Application Services are intended to include all types of application services, whether they are complete proprietary or packaged applications, or whether they are loosely coupled sets of application services.

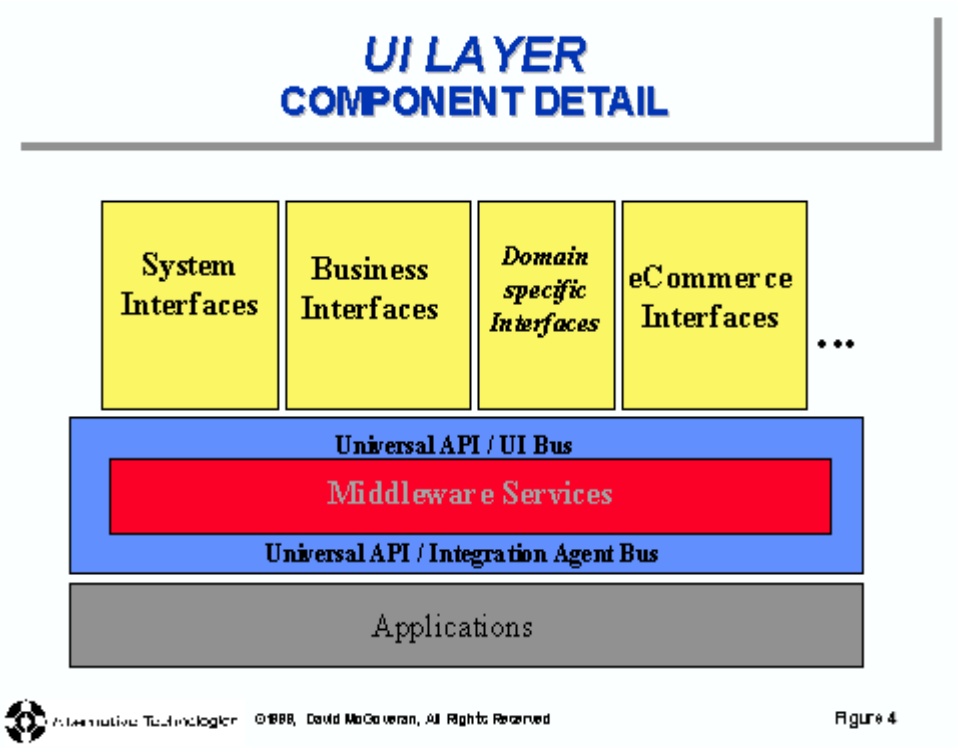

Figure 4

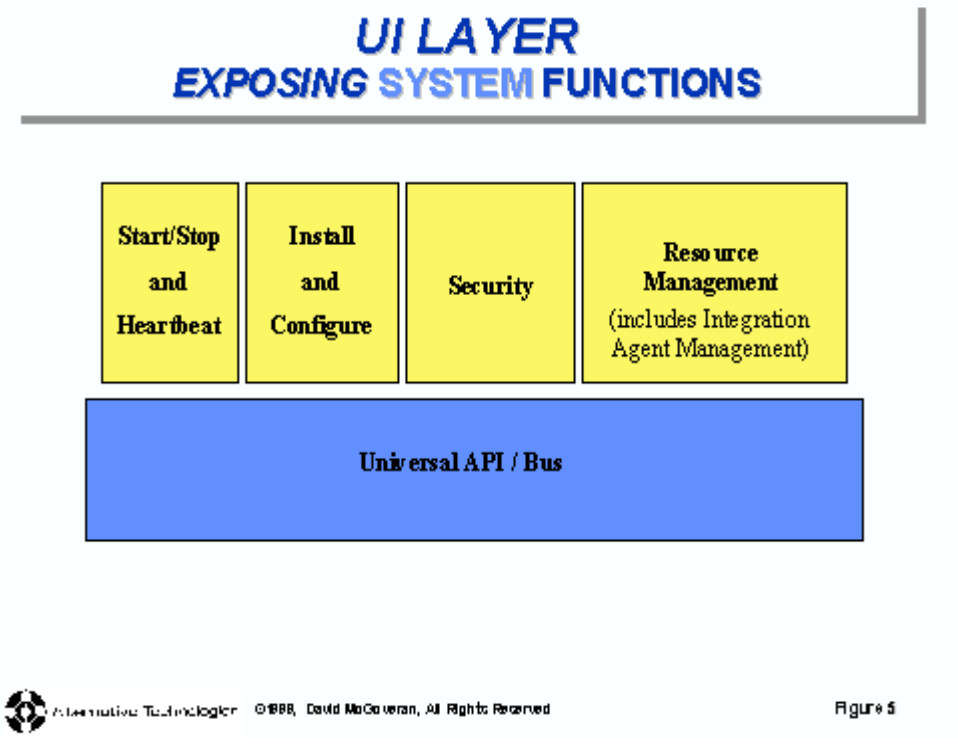

Figure 5

The UI Layer consists of various types of presentation functions (see Figure 4). These include System Interfaces, Business Interfaces, Domain Specific Interfaces, e-Commerce interfaces, and so on. Drilling down on System Interfaces (see Figure 5) we see some (though not all) of the types of specific functions that should be accessible. Drilling down on Business Interfaces (see Figure 6) we see some of the types of business functions that should be accessible including the ability to define and maintain business processes, simulate processes, request inference engine functions, request and view reports, and request and view analyses. Obviously, presentation functions for additional functions are also required such as those for defining data messages between tasks, accessing email services, and so on. What is most clearly implied by these diagrams is that all elements of the UI (presentation) Layer conceptually use a single set of software interfaces. Even if the set of software interfaces consists of multiple physical software interfaces rather than the ideal single consistent set, they are conceptual a single set at this level of the architecture.

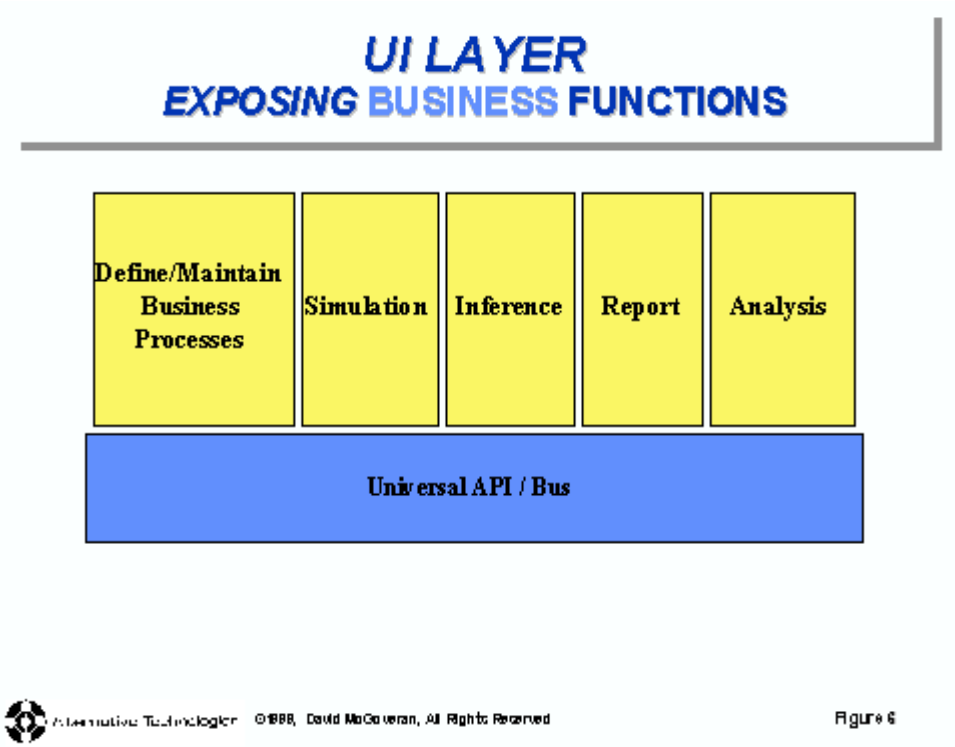

Figure 6

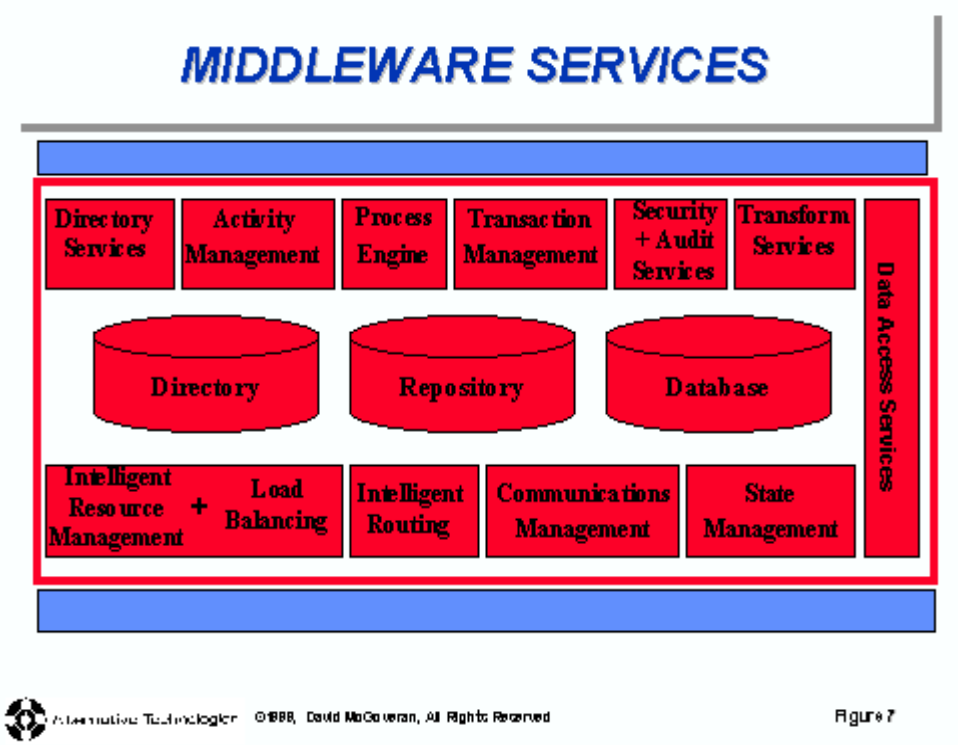

Figure 7

The next diagram describes some of the types of Middleware Services (see Figure 7) that are accessible via the Universal API and is at the same level as that shown in Figure 4. Note that the only block boundary shown is actually the outer red line and that no direct interfaces between specific Middleware Services have been implied. Indeed, I have chosen to show data store related services without the traditional rectangles. Rather, every service is a peer with every other. If interaction is required between two middleware services, they must do so with the Universal API as a mediator. Thus, the structure of the Universal API becomes quite important. I have deliberately chosen not to provide a lower level diagram of Middleware Services because this is the very essence of differences in specific EAI implementations. For example, a message broker might mediate all other service-to-service interfaces, in which case it would be a central block in such a diagram of lower level functionality. Additionally, direct access to some services such as directory services might be the sole province of the message broker, with all other services doing so implicitly through the message broker. In future versions of the EAI Technical Reference Architecture, we hope to provide reference diagrams for various EAI approaches such as a message broker centric approach.

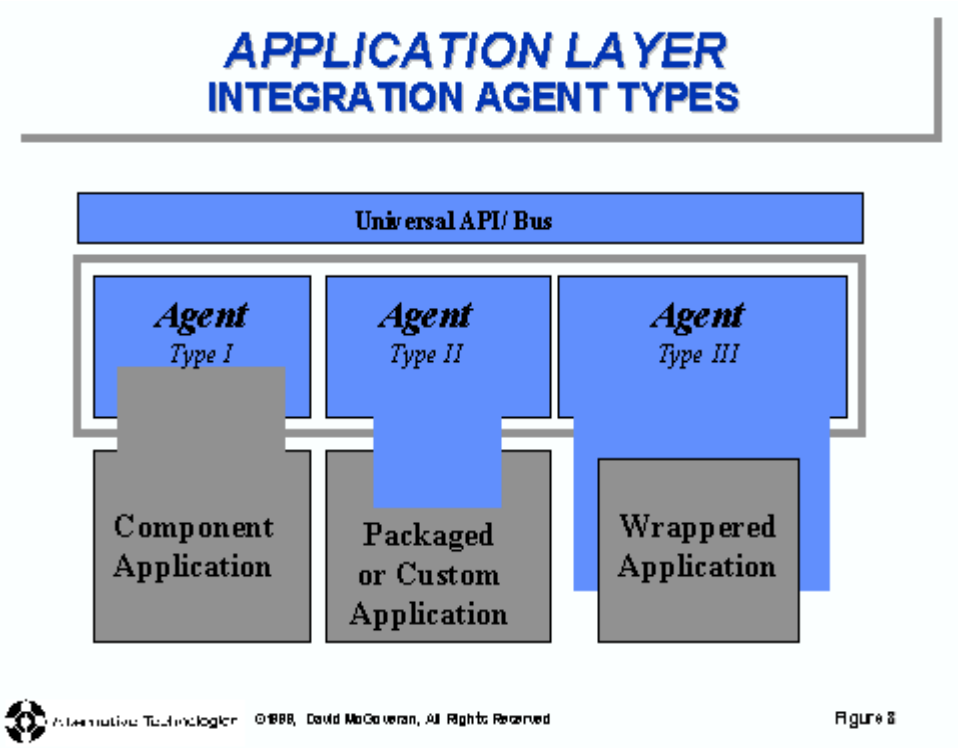

Figure 8

The Application Layer at the second layer of detail (see Figure 8) shows the relationship between Integration Agents and the Universal API. One can think of Integration Agents as extensions of the Universal API, bringing the native interfaces to Applications and Middleware Services in compliance with a thin (i.e., ideal) subset of the Universal API. As can be seen, Integration Agents are broadly of three types. Type I is ideally suited to component based (for example, EJB based) applications in which a component implements an encapsulated business function or task. A Type I Integration Agent provides the means of component integration, as the components have no such capability in and of themselves. A Type II Integration Agent is ideally suited to applications for which a native API exists, as if often the case with packaged applications. Even applications for which the only interface is via the native database fit into this category. A Type III Integration Agent has a complex extension commonly known as a "wrapper" which mediates all access to an application that was not developed with integration in mind, as is often the case with legacy applications.

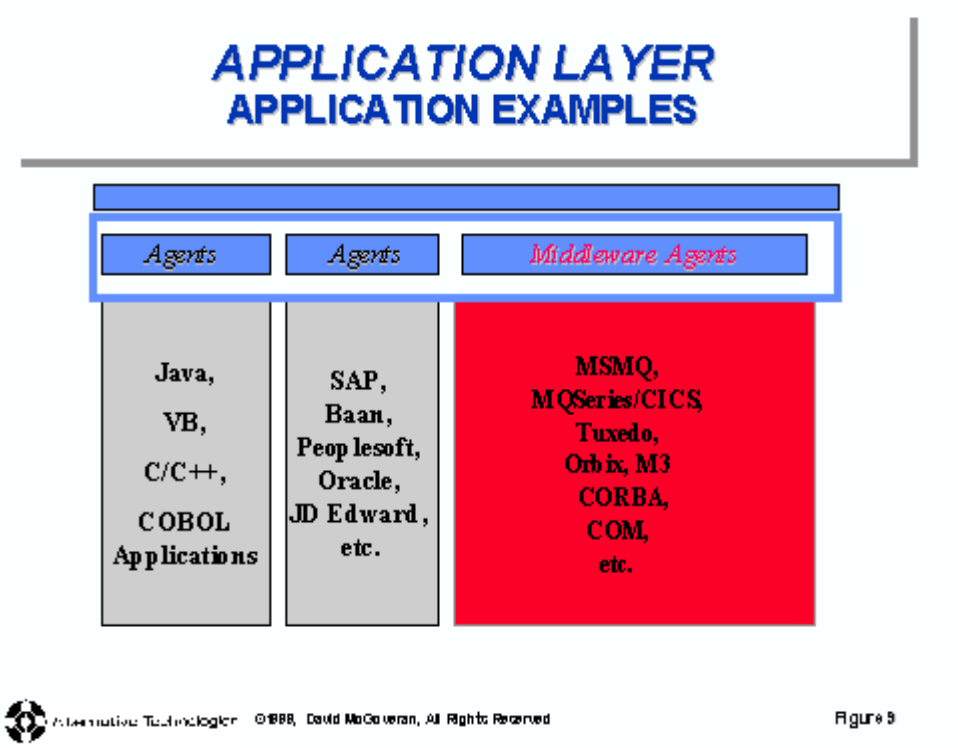

Figure 9

Integration Agents need not provide access exclusively to applications (see Figure 9). Just as the native interfaces of Application Services need to be brought into compliance with an ideal Universal API, so do Middleware Services. A further classification of Integration Agents will be provided in a subsequent version of Alternative Technologies' EAI Technical Reference Architecture.

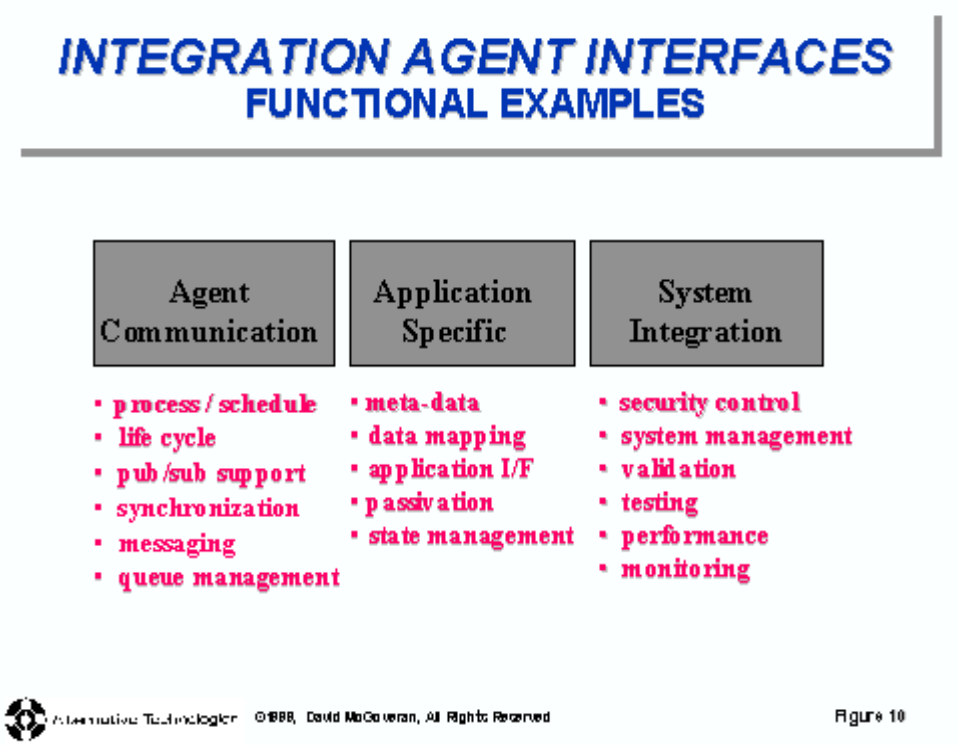

Figure 10

Last but not least, Integration Agents should provide a broad range of APIs (see Figure 10 for some of those that should be provided) if it is to be an effective tool for integration. These APIs should, of course, be documented by Integration Agent vendors. Roughly speaking, these APIs can be classified into those that pertain to Integration Agent Control and Communication, those that are Application Specific, and those that support System Management and Integration. Again, I have deliberately not provided a functional block diagram as the lower level details are implementation (and product) specific.

4. Conclusions

As noted above, Alternative Technologies' anticipates providing future versions of the EAI Technical Reference Architecture in the future. Additionally, as work with specific vendors permits, we will publish specific implementations of portions of the EAI Technical Reference Architecture. For example, we hope to be able to document specific EAI products providing Integration Agents and Middleware Services in terms of the EAI Technical Reference Architecture.

I hope that you, our readers, will find this document useful in trying to understand the diversity that is EAI technology and services. You may it useful for evaluating previous integration efforts, planning new ones, or understanding the strengths and weakness of the approaches of vendors and systems integrators. Your feedback regarding its utility, suggestions you may have for improving it, and case studies regarding its viability are welcome.

Alternative Technologies offers consulting services and seminars on how to use the EAI Technical Reference Architecture for evaluation, complexity comparisons, and planning purposes. For further information, please contact me directly by telephone at 831.338.4621 or email at mcgoveran@AlternativeTech.com.