Isolating Mechanisms in COTS-based Systems

M. T. Gamble R. F. Gamble
Department of Mathematical and Computer Sciences
University of Tulsa
gamble@utulsa.edu

Abstract

Software composition relies on the interaction of software components, either new or sourced from COTS software vendors. Successful component interaction assumes interoperability, which requires the sharing of common characteristics. Shared characteristics make "like" components integrate more easily but also create isolating mechanisms that prevent or inhibit interaction with components that lack such characteristics. Designing systems from COTS components requires understanding their inherent isolating mechanisms. Most approaches only examine composed systems from the perspective of their integration properties. Ignoring or suppressing isolating mechanisms can lead to a partially composed system. Problems may surface anywhere in the system lifecycle including late binding situations at runtime. We examine how shared properties both facilitate membership in a population of like components; yet also serve to inhibit interaction with non-like components. Isolating mechanisms are shown to act across many system levels in the context of COTS based systems.

1. Introduction

Custom-built applications within the same organization tend to follow a well known set of design rules [1]. With each new application, there is very little variance from previous practice and tools. This uniformity, or "likeness," across applications makes integration easier since unexpected interactions rarely occur. The use of COTS software introduces components that are not derived from this uniform heritage and, therefore, are often incompatible in unexpected ways.

Many system integrators choose COTS software based on it having the same underlying technology framework as other components in the desired solution[2]. This choice is driven by an implied heuristic that it is easier to overcome functional discrepancies during integration than non-functional ones imposed by differences in technology frameworks.

It is not practical for a COTS vendor to build their entire product from scratch. They instead make use of other COTS software as a *platform* on which to develop their own products. System developers also rely on COTS software platforms, especially those that represent runtime containers for system deployment such as the Microsoft .NET framework, Java 2 Enterprise Edition (J2EE) application servers or open source LAMP (Linux, Apache, MySQL, and PHP/Perl) platforms.

COTS platform vendors vie for dominance of their software "stack" as the preferred foundation for use by both other COTS component vendors as well as application system developers that ultimately utilize and compose components into complete applications (see Table 1). Control of the underlying software stack represents a significant market advantage to the COTS vendor[3].

Selection of a COTS component that has one of the framework stacks embedded inside establishes a commitment by the system integrator to also use (at least a portion of) that stack. These design commitments are often overlooked and simply implied with the adoption of the specific platform. However, when it comes time to add new components to the system, these earlier platform commitments may resurface and cause integration problems.

Components built on top of a common platform share properties of the platform with other components that utilize it. This is even true of components that were built by different organizations or sourced from different providers. The similarities imposed by the underlying platform become implicit, aiding compatibility across the entire population of components using it[4, 5]. "Like" systems are normally compatible with "like" systems. Conversely, when components are built using differing platforms, there

Table 1. Software platform stacks from IBM and JBoss.

Framework/Stack Function	IBM	JBoss
Web Server	IBM HTTP Server	Apache Tomcat
Application Server	WebSphere App Server	JBoss Application Server
Object/Relational Persistence	DB2	Hibernate
Portal Platform	WebSphere Portal	JBoss Portal
Business Process Management	WebSphere Process Server	jBPM
Business Rules	3rd Party (ILOG, Drools, …)	JBoss Rules
Distributed Transaction Management	CICS, TXSeries	JBoss Transactions
Messaging	WebSphere MQ w/ JMS	JBoss Messaging
Development Tools	WebSphere App Developer, Eclipse	JBoss Eclipse IDE
Security	WebSphere Security, Tivoli Access Manager	JBossSX & Portal Policy Service (JACC based)
Web Services	WebSphere ESB	JBossWS, JBossESB
Connectors/Adapters	WebSphere Adapters	JBoss JCA

are compatibility issues that surface due to differences in the underlying design rules imposed by platforms. These incompatibilities reduce the chances of forming the desired hybrid system.

In this paper we explore the concept of isolating mechanisms in the context of COTS integration. Specifically, our interest lies in those isolating mechanisms that are introduced through the use of software "stacks" from platform vendors both by other COTS vendors as well as by users of COTS products. The assumption is that the properties of isolating mechanisms promote integration in components using the same platform while at the time inhibiting integration with components which use other platforms. We demonstrate anecdotally and formally how likeness leads to isolation as dictated by component or platform isolating mechanisms. Levels of isolation are explicitly defined along with types of isolation and example of isolating mechanisms common to each level. A better understanding of these properties will facilitate a more informed design inquiry during the COTS evaluation and selection process.

2. "Likeness" leads to isolation

Following an analogy from biology [6, 7], we believe that isolating mechanisms in software systems serve dual purposes. They both promote integrations among similar components and inhibit them in dissimilar components. Commercial technology platforms can induce similarities across all components that use them. Most of these platforms include both industry standard open interfaces as well as some amount of proprietary functionality. For components using the platforms, it is this proprietary functionality that creates couplings between the components and the framework, both making it easier for the component to interact with other components that share the proprietary functions and harder with those that do not. The proprietary functions of the platform, shared by all components based on it, act as isolating mechanisms.

Consider the two off-the-shelf platform stacks in Table 1. Each is based on a set of Java standards, such as Java 2 Enterprise Edition (J2EE), which makes them alike in some ways. Each understands and can execute Java language programs. Each can communicate using the RMI distributed object protocol. However, each stack also includes implementations which are unique to the stack. For example, even though both the IBM Websphere and the JBoss application servers support execution of industry standard Enterprise Java Beans (EJBs), they have unique tooling for deployment and monitoring of those EJBs. For applications built within a specific stack, they benefit from a tight integration with these unique, or proprietary, features. Where both WebSphere MQ and JBoss Messaging support a standards Java Message Service (JMS) API, they each implement queuing logic in different ways. The vendors in many cases provide proprietary extensions to the industry APIs in order to add distinguishing features.

Selection of a COTS component that has one of the framework stacks embedded inside establishes a commitment by the system integrator to also use (at least a portion of) that stack. These design commitments are often overlooked and simply implied by the adoption of the specific platform. However, when it comes time to add new components to the

system, these earlier platform commitments may resurface.

2.1. Recognizing isolation

The recognition of an isolating mechanism starts with understanding a problematic integration. By definition, components which share the same isolating mechanism will not experience integration issues due to those isolating mechanisms since they are inherently compatible.

Consider a scenario where it is desirable to extend an existing software system by adding a COTS component. The existing system includes components for the business applications of a Web portal for *ordering* and an application for order *fulfillment*. The system owner wants to extend these business services by adding a *billing* component that supports online electronic bill presentment and payment (EBPP) through the existing portal as in **Error! Reference source not found.**.

The existing system (*ordering* + *fulfillment*) was built by the system owner using products from the IBM supplied COTS platform (refer to Figure 0).

The *billing* component is to be sourced from a COTS supplier that has a family of products built on top of JBoss. Initially, the integration appears straightforward. The existing application contains a COTS portal, namely IBM's WebSphere Portal server. It provides a means to access portal content structures (portlets) remotely through an industry standard application programming interface (API) protocol called Web Services for Remote Portlets (WSRP [8]). That same protocol is supported by the JBoss Portal

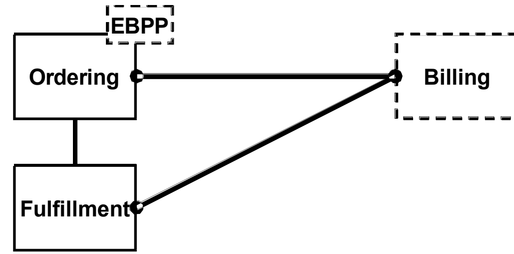

Figure 1. Desired composite application.

Server in the billing component which can serve up a portlet containing the EBPP function to the ordering component at runtime. Details of the integrated system are shown in Figure 0.

The integration of the ordering portal and the billing portal satisfies the need to reveal the billing functions in the existing portal. WSRP has a facility to pass authentication credentials, but not authorization data. The expectation is that the service publisher (in this case JBoss Portal) will implement access control.

The JBoss Portal product includes features for managing authorization data and access control based on that data. These features are implemented using a standardized set of Java classes called Java Authorization Contract for Containers (JACC). JACC is a set of security contracts defined for the application containers (such as portals and Java application servers). The containers use JACC in the server to restrict client access to resources and services. Use of the JACC library and JBoss's implementation of a local instance of a JACC data store make it easy to integrate across multiple applications using the same

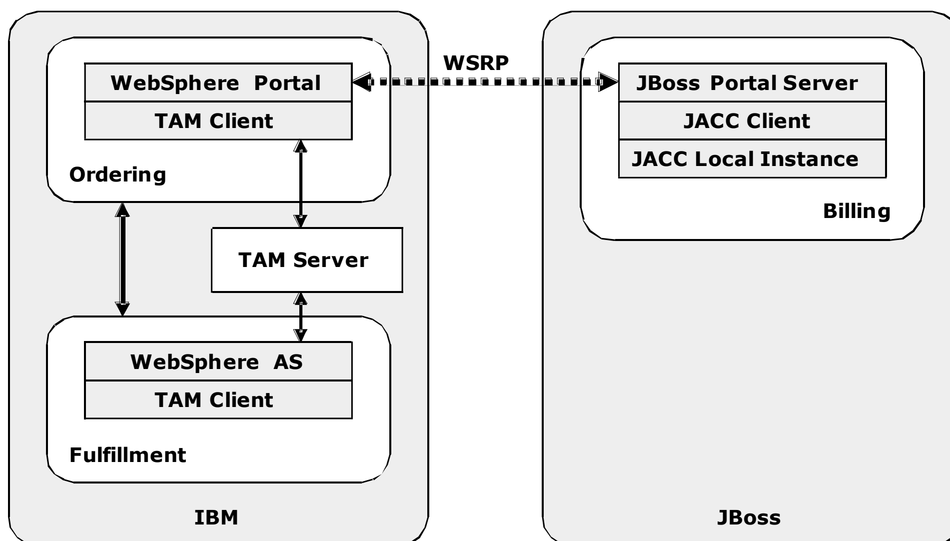

Figure 0. Component Populations Built on IBM and JBoss Stacks.

portal engine.

The main problem is that the authorization data in JBoss is not visible outside the environment and WSRP provides no way to communicate authorization data between the IBM and JBoss-based component populations. Added to the problem is that the IBM components share a common security service, IBM's Tivoli Access Manager (TAM) product, which allows authorization data to be shared across all applications incorporating TAM access control agents.

The resulting system, a composition of the *ordering*, *fulfillment* and *billing* components supports interactions for displaying billing information via a portlet through the original ordering portal. Authorizations to click on the EBPP service are controlled by the IBM WebSphere Portal. The portlet service, which provides the EBPP interface, is controlled through authorizations in the JACC implementation on the JBoss platform. Thus, it is possible, and highly likely, that this authorization data will become out of synch or subject to different policies. In that case, a user might be able to select online bill payment when interacting with the Ordering component on WebSphere, only to have the propagated service response denied by the Billing component on JBoss. This is clearly an undesirable effect of the integration. Worse, it can escape detection until after deployment when the authorization data changes.

2.2. Modeling the isolation

Formal modeling allows an unambiguous representation of component functions, both public and private, as well as platform and integrated system requirements. Though examining isolation from such a detailed component perspective is desirable, it is necessary to first understand it at a higher level so that future modeling attempts can convey the important aspects. In this section, we illustrate that if modeled properly, the isolation is manifested in the example system in Section 2.1.

Let *UIDs* be the set of all possible user identifiers. Let *Actions* be the set of all possible actions for components. We define S as a relation on users and actions with respective authorizations.

$$S: (UIDs \times Actions) \times Boolean$$

S is a generic relation expected to have some bearing on the integrated system. Each component on the IBM stack *I-Stack* shares a global authorization function. Thus, $IAuth \subseteq S$, but is more restrictive than S, as follows

$$IAuth: (UIDs \times Actions) \rightarrow Boolean$$

Each ordered pair of *UID* and *Action* maps to either *true* or *false*. For all $I_n \in$ *I-Stack*, only authorized actions can occur. This is accomplished through the access control mechanism of having each component of *I-Stack* guard invocations of its local actions with validations against *IAuth* as follows:

$$\text{Given } (u,k) \in (UIDs \times Actions) \text{ Perform } k \text{ as } u \text{ if } IAuth(u,k)$$

The function *IAuth* is the same for all $I_n \in$ *I-Stack*, meaning they are subject to the same authorization controls. During composition, components of *I-Stack* easily mesh with respect to authorizations since they all share a common implementation.

Let *J-Stack* be based on the JBoss stack. Each $J_m \in$ *J-Stack* has its own local authorization function, *JAuth*, which is similar to *IAuth*. The difference is that each $J_m \in$ *J-Stack* is responsible for its own authorization data. This means that the authorization function for the JBoss stack must be defined in terms of each $J_m \in$ *J-Stack*. That is, it cannot be globally shared.

We define JAuth as follows.

$$JAuth: J\text{-}Stack \rightarrow (UIDs \times Actions) \rightarrow Boolean$$

In *JAuth*, each component is mapped to its own authorization data. For simplicity of the example, we assumed that identities are still global, so *UIDs* are compatible across all *J-Stack* components.

Given that the *ordering* component in Section 2.1 is assumed to be the new entry point for users into the combined system, we must guarantee that users who are authorized to select a *billing* action that is now visible in the *ordering* portal are also authorized to execute that action when the request is processed by the *billing* component (via the remote portlet invocation).

The following *invariant* cannot be violated.

$$\text{For all } u \in UIDs, k \in Actions: IAuth(u,k) \Rightarrow JAuth(j,u,k) \text{ for some } j \in J\text{-}Stack$$

Given that the two functions *IAuth* and *JAuth* are not equivalent, the invariant cannot be guaranteed because there may not exist the necessary $j \in$ *J-Stack*. Recall that elements of *JAuth* are 4-tuples that include a component from *J-Stack*. *IAuth* has no such constraints. Even if, by circumstance, the authorization data is the same across *I-Stack* and *J-Stack* components, we have no assurance of that

Table 2. Types of isolation in components

INTEGRATION PHASE		ISOLATION TYPE	DESCRIPTION
Pre-integration	Pre-message transfer	Spatial	No interconnection network between components
		Designation	Components are not within the same name space and are not visible to one another.
		Temporal	Components cannot synchronize. They are not concurrently “active”.
	Post-message transfer	Syntax	Format of public interfaces aren’t compatible (e.g., type or order of invocation parameters differ). The message cannot be parsed.
		Protocol	Behaviors are incompatible at the interface level. For example, the messages are out of sequence.
		Semantics	Meaning of interface types or message payloads is inconsistent between components. The message cannot be understood.
Post-integration	1st Hybrid	Inviability	The resulting hybrid isn’t robust and doesn’t persist until a composition occurs
		Sterility	The resulting hybrid cannot be further composed with any other components
	2nd Hybrid	Breakdown	The hybrid may be composed but the result will be weak and subject to inviability and/or sterility

because of the localization of the data in the *J-Stack* components.

3. Isolation levels and types

Designing systems from COTS components requires understanding their inherent isolating mechanisms. There are numerous ways in which components can be incompatible at the interface level, but integration challenges extend further than interfaces, spanning the entire lifecycle of integration. Such integration challenges, if not overcome, result in isolation. To overcome these challenges, system integrators must identify and “defeat” the isolating mechanisms which are acting to cause isolation.

Difficulties arise when trying to classify isolation types in order to make the discovery of isolating mechanisms confined to the scope of the isolation in question. Table 2 presents isolation types within a framework of an integration lifecycle, classifying the isolation type by the kind of isolation imposed and the phase in the integration lifecycle where isolating mechanisms act to make it happen. This approach is based on a similar classification in biological evolution theory [6], but recast into a software context. There are two main categories, pre and post-integration.

In the first category, integration fails outright due to extreme incompatibilities that occur. An isolating mechanism may exist to thwart transmission of a message by lack of connection (spatial isolation). This can occur due to lack of a physical pathway through a network or it could be caused by a firewall or network admission control device which separates the component from others. Even though compatible components are connected, they may be unable to identify other components for interaction (isolation due to failures in designation or naming). One such occurrence is when components have been unable to register with network naming services or have been incorrectly registered. Finally, messages may never be transmitted between components because they aren’t synchronized to be available at the same time.

Once a message is transferred it can still fail to be used because

- The syntax of message is incorrect. A common cause is version mismatch. One component has received an update to interface conventions while the other has not.
- The protocol (behavior) required by one component is different than another. For example, certain messages may have preconditions (e.g., “capture” a token message before transmitting, or process an authentication before accepting a request).
- The meaning of the message is unclear (semantic isolation). Common causes are lack of coordination across components on common terms

or data structures which are assumed by the message.

If a message is received and understood, it is presumed that a form of integration has been accomplished. An interaction, at least at the interface level is complete. If this is the case, and the interaction was between components which were previously not of the same kind (e.g., they were based on different platforms), then a *hybrid* is formed. Hybrid components are simply compositions of two (or more) unlike components. From the perspective of the systems integrator of COTS components, almost all integration activities form hybrids. The likelihood that a COTS component is completely "like" internally developed components is remarkably low. However, most COTS components include tools and design guidance to assist integrators in overcoming isolating mechanisms acting in the pre-integration phase. Such tools include adapter toolkits that can convert between interfaces with incompatible syntax, protocol or semantics.

Given that the COTS integrator's primary work is now left to forming "good" hybrids, it is important that it is understood how post-integration isolating mechanisms may reduce the viability or fertility of hybrids or their progeny. A robust hybrid component must be formed by means of composition from unlike components, where that "unlikeness" is induced by platform differences. The robustness of the resulting hybrid is determined by its *viability* (capability to achieve further valid execution states) and its *fertility* (ability to participate in further compositions). Clearly, the robustness of the hybrid should be of particular interest to COTS system integrators in determining a successful integration. With proper specification, compositions can be analyzed for these properties.

Returning to our earlier example where components were based on either IBM's software stack or JBoss's, the resulting hybrid was found to be *inviable*. We could not assure that further execution states of the hybrid would be valid. The inconsistent control mechanisms for authorization data established a situation where a valid action in one component could be invalid in the other, thus breaking the composition.

4. Relevant Research

In this section, we cover a variety of research that contributes to our terminology and findings.

4.1 Isolating Mechanisms in Biology

The first use of the term *isolating mechanism* was by Dobzhansky [6] in the context of defining biological species: "Species are systems of populations: the gene exchange between these systems is limited or prevented by a reproductive isolating mechanism or perhaps by a combination of several such mechanisms." The key was that isolating mechanisms impair gene exchange between different species. Mayr [7, 9] adopted the use of the term and applied it to what became the "biological species concept".

> "Among the attributes members of a species share, the only ones that are of crucial significance for the species definition are those which serve the biological purpose of the species, that is, the protection of a harmonious gene pool. These attributes were named by Dobzhansky *isolating mechanisms*."[9]

Mayr goes further to say that isolating mechanisms act at the level of individuals, meaning that the mechanisms must actually occur in a single organism and act between that organism and another. This distributed control necessary for protecting system level properties (i.e., "a harmonious gene pool") echoes modern design directions in computing. Perimeter based security designs in computing networks have moved towards a more distributed philosophy of "defense in depth" where access control is collocated with or within the asset being protected.

Also, in biology, isolating mechanisms can be "leaky," meaning that they are not able to completely enforce separation. However, the idea is that even though particular pairs of individuals from differing species may be able to reproduce (i.e., forming a hybrid), the overall populations of individuals in the systems remain separated. Complete species "fusion" is not allowed. In many cases hybrids are impaired in some way, either being short lived (inviable) or unable to reproduce (sterile).

Following the biological analogy leads to a determination that isolating mechanisms in software systems are those properties which serve to protect the interoperability and cohesion of components within a population. Depending on perspective, such isolating mechanisms can be viewed in two ways. They both facilitate the interaction with conspecific (of the same species) components as well as prevent the integration with those which are non-conspecific. Conversely, and following the biology analogy, isolating mechanisms may be "leaky" in that they simply inhibit integration, but do not fully eliminate it, between all conspecific components.

4.2 Isolating Mechanisms in Business

Economic analysis and theories of a resource-based view (RBV) of the firm have borrowed the term isolating mechanisms from biology. Rumelt [10, 11] introduced use of the term in an economic context to describe those mechanisms which create barriers to competitors' ability to duplicate resources. However, it has been pointed out by Fahy [12], that some of the barriers identified by Rumelt are not isolating mechanisms since they describe economic market conditions that are external to the firm.

A further formulation is made by Norton [13]; isolating mechanisms are the devices of a firm which protect rent (i.e., profit) generating resources from imitation. Thus, isolating mechanisms in an economic context form a basis for maintenance of competitive advantage. This view aligns well with the biological term. From a natural selection view, any species will preserve those traits which both distinguish it from other species and provide it advantage during selection. It can also be argued that, since the uniqueness derived from proprietary software components provides competitive advantage to component suppliers, isolating mechanisms in software components can be a direct realization of an economic isolating mechanism acting to preserve software-based resources from imitation in the component market. The incompatibilities, enforced through isolating mechanism of proprietary components, are advantageous to the supplier when they create barriers to entry and substitution.

4.3 Adapters create integration hybrids

Integration approaches that rely on adapters or bridges focus on making a non-compatible functional interface appear to be compatible; effectively projecting the interface of a component based on one platform into a form that is compatible with another target platform. This approach is greatly favor
ed by current vendors of integration products for service-oriented architecture (SOA) development and operations environments. These products include various forms of enterprise service bus (ESB) implementations and large adapter "suites." One such vendor includes hundreds of adapters it what is calls a "universal adapter" [14]. Each is fitted to an existing, proprietary interface set from some other vendor to a common representation in either a language API (such as Java) or normalized protocol (such as a Web services interface).

5. Conclusion and Future Work

Lack of understanding for the causes of isolation impacts COTS-based system integration, including its reusability, cost of adaptation, and evolution. As yet, no processes exist for discovering isolation mechanisms in COTS-based systems past the interface level at a post-integration phase. Given that COTS integration is primarily an exercise in constructing hybrid components, this post-integration phase is crucial to assuring that integrations are durable and robust. Discovery and analysis of isolating mechanisms by COTS integrators can improve integration success as well as inform the evaluation of COTS tools.

We believe the process by which isolating mechanisms are discovered in particular systems can proceed by applying known analysis techniques across the various integration phases and isolation types. For static integrations, analysis can be done on actual design specifications. For dynamic integrations, analysis occurs over a range of use cases which represent dynamic integration scenarios (perhaps through simulations). Essential research in definition, demonstration, and taxonomies of classification, such as presented in this paper, must precede discovery. Formal modeling can clearly capture isolation, but extensions to modeling languages are necessary to be able to consistently express isolation as safety and progress properties over the composite functions of the COTS components. Because formalisms also contribute to the formulation of a discovery process, a larger modeling effort is the next step in our isolation research.

Acknowledgement. This work is supported in part by a U.S. AFOSR award FA9550-05-1-0374. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

6. References

[1] C. Y. Baldwin and K. B. Clark, *Design Rules, Volume 1: The Power of Modularity*. Cambridge, MA: MIT Press, 2000.

[2] J. Li, R. Conradi, O. P. N. Slyngstad, C. Bunse, U. Khan, M. Torchiano, and M. Morisio, "Validation of New Theses on Off-The-Shelf Component Based Development," in Proceedings of the 11th IEEE International Metrics Symposium (Metrics'05), Como, Italy, 2005.

[3] M. LaMonica, "Software's 'stack wars'," CNET News, 2006.

[4] F. Buschmann, R. Meunier, H. Rohnert, and P. S. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*: Wiley and Sons Ltd., 1996.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison Wesley, 1994.

[6] T. Dobzhansky, *Genetics and the Origin of Species*, 3rd ed. New York: Columbia University Press, 1951.

[7] E. Mayr, "What is a species, and what is not?," *Philiosophy of Science*, vol. 63, pp. 262-277, 1996.

[8] A. Kropp, C. Leue, and R. Thompson, "Web Services for Remote Portlets Specification, v1.0," OASIS, 2003.

[9] E. Mayr, *What Evolution Is*: Basic Books, 2001.

[10] R. P. Rumelt, "Towards a strategic theory of the firm," in *Competitive Strategic Management*, R. B. Lamb, Ed. Engelwood Cliffs, NJ: Prentice-Hall, 1984, pp. 566-570.

[11] R. P. Rumelt, "Theory, strategy and entrepreneurship," in *The competitive challenge: Strategies for industrial innovation and renewal*, D. J. Teece, Ed. Cambridge, MA: Ballinger Publishing Company, 1987, pp. 137-158.

[12] J. Fahy and A. Smithee, "Strategic Marketing and the Resource Based View of the Firm," *Academy of Marketing Science Review*, 1999.

[13] B. Norton, "Isolating Mechanisms: Can Managers Protect Rent Generating, Knowledge Based Assets?," in *Proceedings of the USASBE/SBIDA 2001 National Conference*. Orlando, FL, 2001.

[14] R. Scherwin and J. Freivald, "Reusable Adapters: The Foundation of Service-Oriented Architecture," iWay Software, 2005.