Sun microsystems

SYSTEMATIC DEVELOPMENT AND THE SERVICE ORIENTED ARCHITECTURE

White Paper
March 2006

Table of Contents

Introduction 2

The Birth of Composite Applications 2

Composite Applications and Service Oriented Architecture 4

Systematic Development of Composite Applications 5

Modularity in the Service Oriented Architecture 8

The Technology Required to Build an SOA 11

Introduction

One of the ongoing challenges in IT is managing the cost, time, and risk associated with application development. Regardless of application type, IT has a reputation for blown schedules and budgets — and that is when applications actually make it into production. In a recent Standish Group survey, only 16 percent of software projects were completed on time, within budget and to specification.

Addressing this miserable track record is critical. Yet, so many panaceas have been proposed over the years, it is difficult to separate the real from the imagined.

In the end, most thoughtful IT managers and analysts suggest that there are three fundamental tenets of effective application development and delivery. First, development has to follow a well-defined methodology — one that fosters continually improving adherence to best-of-breed development processes. This is the basic message of the Software Engineering Institute's Capability Maturity Model.

The second key to development mastery is a mature governance model. By involving all of the stakeholders in decision making at every step of the development process, you can ensure support for the organizational development challenges that are central to building applications that meet the real needs of the enterprise. Also, a solid and accepted governance model facilitates the process and functional changes in the business — changes that are necessary to support the implementation of new applications.

The third essential element in building applications that deliver the required functionality on time and on budget is architecture. Specifically, successful application projects require a set of blueprints that provide all parties with a clear, graphical view of the solution components and their interdependencies.

Taken together, methodology, governance, and architecture constitute a systematic approach to development — any kind of development.

The Birth of Composite Applications

A composite application is similar in some ways to the traditional stovepipe or silo applications that IT has been building for the past 40 years. It maintains data, processes logic, and delivers the results to people, devices, or other applications.

The difference has to do with *green fields* and leverage. At this point in the evolution of enterprise applications, IT seldom builds massive green field applications — starting with a blank sheet of paper to build a new database, application logic, user screens, and so on. Instead, most new applications fall into one of two categories. Either IT purchases and implements a packaged application or a new application is built *on top* of existing applications.

This notion of building applications on top of existing applications got its start in the late 1980s with screen scraping. Using programming tools that could interact with mainframe systems as if the new program was a computerized user, developers built screen-scraping PC applications that could read and write to fields on mainframe application screens.

Initially, these screen-scraping applications merely placed more attractive front ends on ugly, hard-to-learn and difficult-to-use mainframe systems. Quickly, though, developers began to employ these tools to create new applications — programs that could pull data from multiple back-end applications while presenting a unified, coherent front end to the user.

At about the same time, the introduction of database access products based on Open Database Connectivity (ODBC) gave developers the ability to write PC code that could read and write to multiple Database Management Software (DBMS) and file systems with the same tool set. Now, they could readily write an application that could access mainframe databases and files along with UNIX® and Microsoft Windows data. And, that is what developers did — they built systems that could combine mainframe, UNIX, and Windows data into one application.

Thus was born the composite application.

Composite Applications and Service Oriented Architecture

But, that does not mean that these composite applications were built well. As with monolithic applications built on a single source of data, composite applications proved to be difficult to deliver on time, within budget, and to specification (see Figure 1). Clearly, a more professional, systematic approach to development was needed.

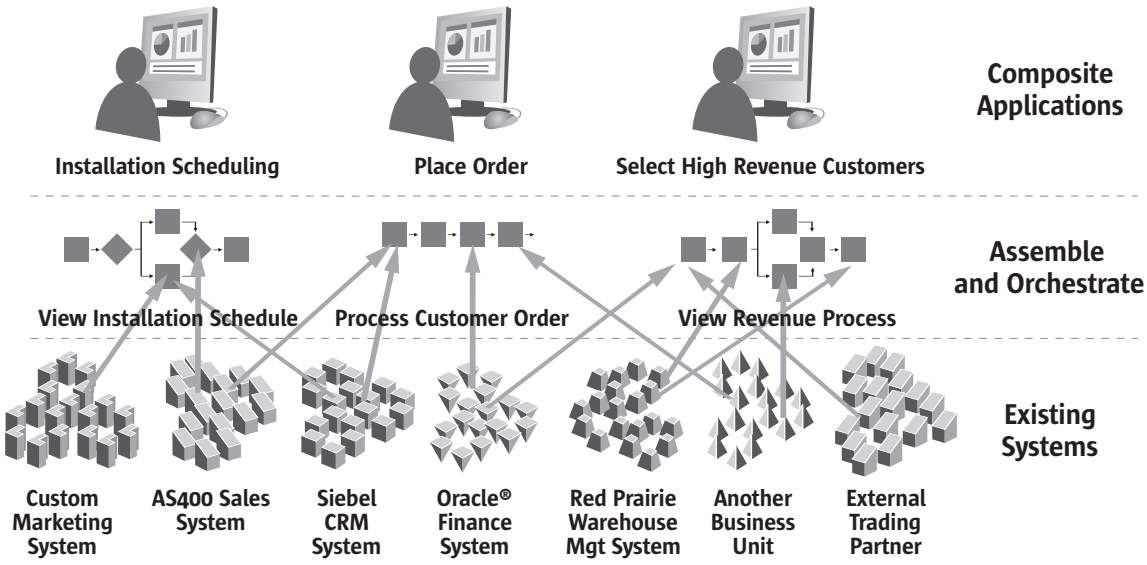

Figure 1. Composite applications have often been built without a coherent architecture

That's where the service oriented architecture (SOA) comes in. An SOA positions software resources as services — discrete business-oriented components that provide functionality while masking the underlying implementation details. In other words, an SOA provides a layer of abstraction that hides the complexity of legacy systems and packaged applications, providing access to their functionality while presenting a location-independent, business-oriented interface for the programs consuming that functionality.

The concept of an SOA is not new. The Common Object Request Broker Architecture (CORBA) and Microsoft's Component Object Model (COM) have long provided SOA-type capabilities. These approaches to service orientation, however, presented developers with challenges that limited SOA deployment.

First, CORBA and COM are tightly coupled, which means that the implementation of the program providing the service is closely linked to the implementation of the program requesting the service — if you change one, you have to change the other. This tight coupling means that changes to either the service provider or consumer are difficult to manage, because everything has to be planned and implemented in a highly coordinated fashion.

Second, CORBA and COM-based SOAs were proprietary. Microsoft controls COM and, while CORBA is standards-based, real-world CORBA architectures are typically built on a single vendor's implementation of the specification. Why? Because each vendor's interpretation of the standard varied just enough to prevent seamless interoperability.

Finally, neither COM nor CORBA (nor, for that matter, .Net, Web services, Enterprise Service Bus or ESB) are, on their own, sufficient to build and deploy an SOA. Other capabilities are needed, including application-to-application (A2A) and business-to-business (B2B) integration, Business Process Management (BPM) and workflow, Business Activity Monitoring (BAM) and complex event processing, single customer view and portal functionality, identity management, Extract, Transform, and Load (ETL) and application server functionality. These technologies form the basis of a complete application development platform that can support applications that leverage an SOA to access a broad array of existing infrastructure components and services.

Systematic Development of Composite Applications

A systematic approach to effective composite application development is based on an SOA (see Figure 2). By employing a repeatable methodology to implement a well-considered architecture within a mature governance model, enterprise IT can ensure the most-cost effective and efficient development projects. Now, that is not to say that it is impossible to build composite applications without an SOA. But the time and effort are higher, the risks of exceeding your budget or schedule are higher, and your ability to reuse the application for related projects is lower.

So, what goes into an SOA and how does that relate to composite application development?

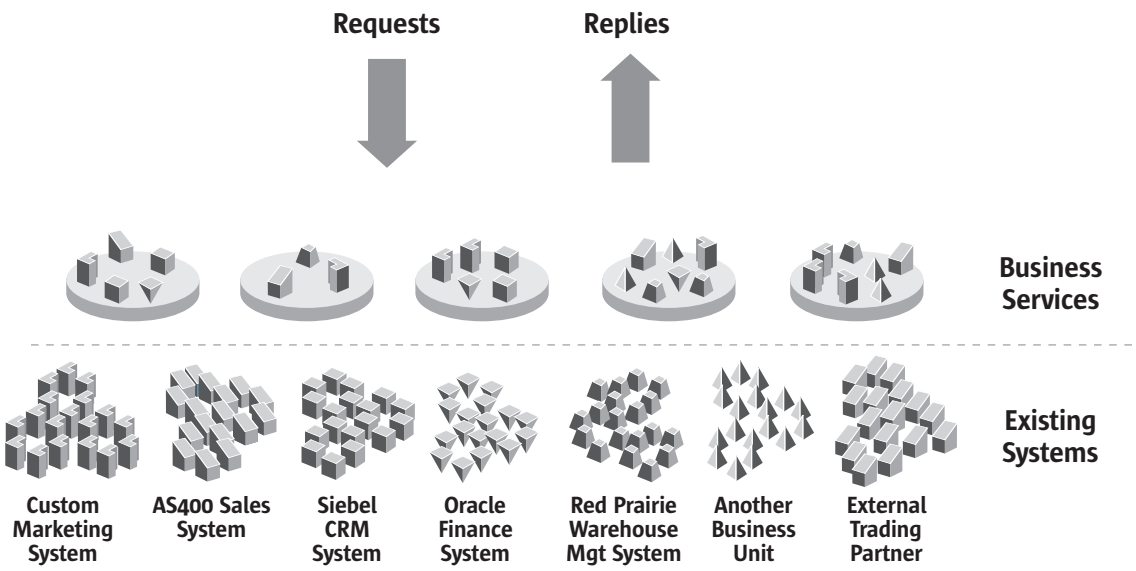

Figure 2. Systematic composite application development relies on a Service Oriented Architecture

First, look at the type of interaction that is required to implement an SOA. Like the client-server applications of 15 years ago, an SOA fundamentally relies on a set of request/reply interactions between a client program (service consumer) and server program (service provider).

This has two useful implications. First, it suggests that the development mindset that began with two-tier, client-server applications and extended with n-tier, Web-based applications is evolving in a natural way to encompass this new architectural paradigm. That is important, because IT is more likely to embrace an evolutionary architecture than an architecture that represents a radical departure from what has gone before.

The second implication is that the SOA is able to readily leverage existing client-server and n-tier applications — since these applications are made up of programs and program components that are already built to respond to client requests.

But, an SOA does not stop there. In order to support a broader set of application requirements, the SOA evolved to include a second mode of program invocation and program-to-program communication — messaging.

Specifically, the SOA is complemented by a similar design pattern, Event Driven Architecture (EDA). Based on asynchronous communication, an EDA allows programs to send messages to various destinations (people, devices, and other programs) without waiting for a response (Figure 3).

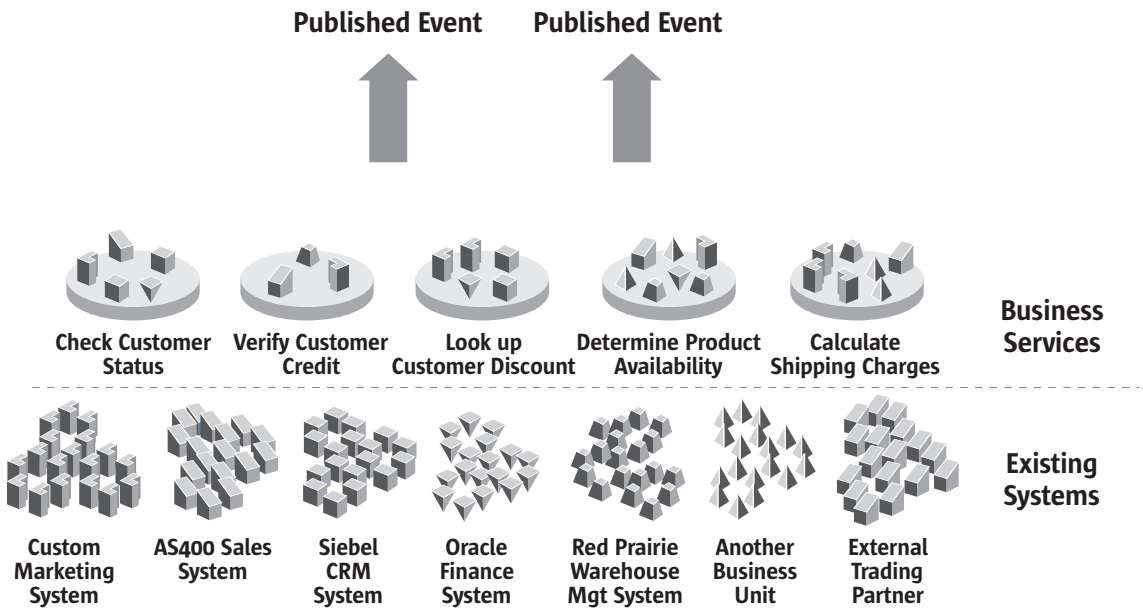

Figure 3. Event Driven Architecture — a pub/sub variant of the Service Oriented Architecture

Two basic addressing methods are used to support the EDA's messaging-based communication. The first is the send/receive method, where a message is sent from one source program to one target program. The sending program does not have to know a specific network address for the target program (that is handled by the messaging middleware), but the two programs do have to agree on the circumstances of the data exchange — When will it occur? What is the content and format of the data that is sent? What security mechanism is used to ensure that the source and target programs are not spoofed? What logging and messaging management functions are used to guarantee that the message is delivered, that only one copy of the message is delivered, and that the messages are delivered in the order in which they were sent?

The second addressing method used in an EDA is the publish/subscribe (pub/sub) model. Like send/receive messaging, both sides of a pub/sub interaction must agree on data semantics, security, and line protocol (that is, the gritty details that govern the exchange of data packets and acknowledgments between sender and receiver). Unlike the deterministic addressing of the send/receive model, this approach provides an abstracted addressing capability that is implemented via a subscription process. The source application provides a message of a specified type to the pub/sub middleware. Applications that want access to messages of that type post a subscription to the middleware. The middleware then provides the intelligence to link published messages to subscribers.

Like send/receive messaging, the sending program in the pub/sub model does not wait for a response from a message's recipient. Unlike send/receive messaging, in pub/sub messaging each message is transmitted without any knowledge on the sender's part as to who or what or even how many will receive the message — there may be no recipients at all, or there may be thousands.

Modularity in the Service Oriented Architecture

As mentioned earlier, an SOA is based on creating an application from software building blocks — discrete, business-oriented components that provide required functionality to the programmer, while masking underlying implementation details.

This notion of modularity offers advantages in any type of construction, whether it is constructing a house, car, or bicycle (Figure 4). In the domain of software and applications, the primary advantage is code reuse — the ability to build a component once, then use it in multiple applications.

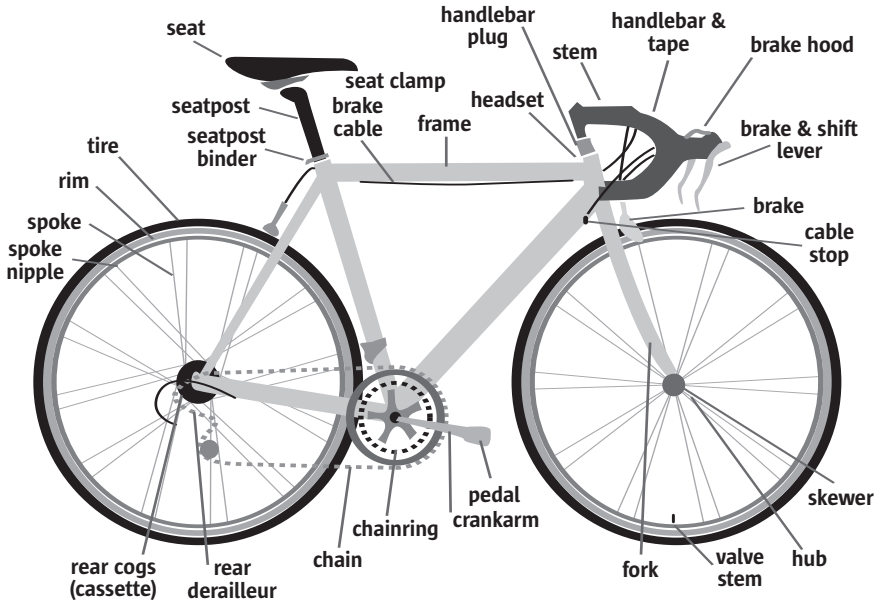

Figure 4. Modularity is an essential tenet of an SOA

On the surface, this is an embarrassingly obvious observation. But a lot goes into building a component-based, application architecture that separates theory from the real world.

For example, when discussing CORBA earlier, we mentioned that the lack of strictly enforced implementations of the CORBA standard hindered the interoperability and portability of CORBA applications. Another weakness of CORBA is the fundamental reliance on Object Oriented Programming and Design (OOPD) within CORBA application development.

OOPD is built on the notion of *objects*. To the ordinary observer, these objects are the same as the services that are used to build an SOA. But there is an important difference. In OOPD, opaque concepts like inheritance, polymorphism, classes, and methods wrap objects in a cloak of mystery.

The SOA, on the other hand, takes a much more business-like approach to IT architecture. In an SOA, the application domain (for example, manufacturing, logistics, HR, sales, or marketing) is broken down into basic business objects such as bills of materials, invoices, customers, and shipments.

With each of these basic business objects, we add an action to create a business service. For example, we might "check customer status," "verify customer credit," "look-up customer discount," "determine product availability," or "calculate shipping charge" (see Figure 5).

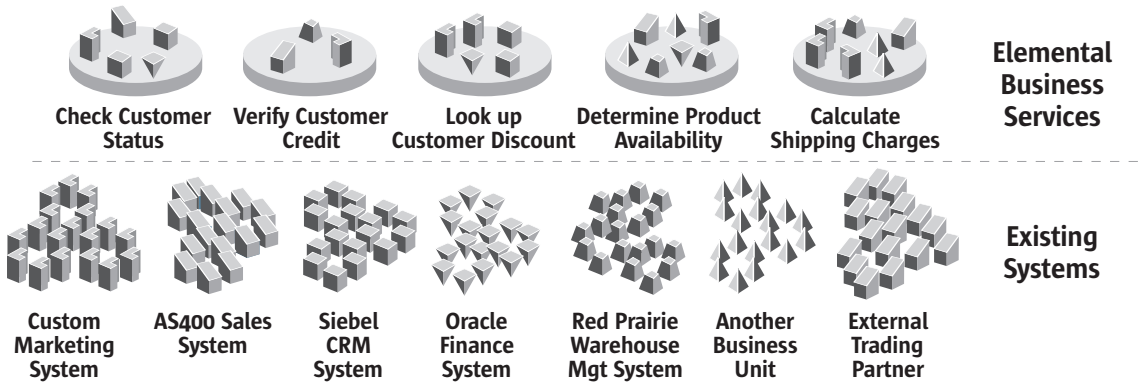

Figure 5. Functional decomposition enables the identification of the components that make up each Line of Business process

At this point, we now have basic business services, the lowest-level building block of an SOA and the fundamental component in the development of a composite application.

From here, elemental business services are combined in various ways to create composed business services. For example, all five elemental business services might be combined to create a *process customer order* business service (Figure 6), while *check customer status* and *determine product availability* might be combined to create an *installation scheduling* business service (Figure 7).

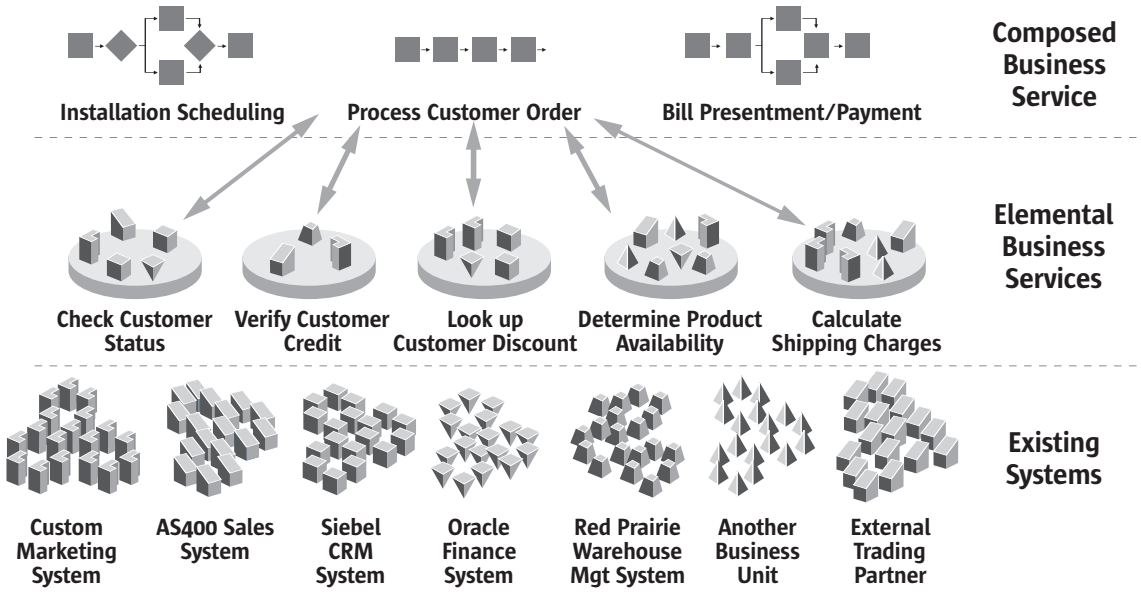

Figure 6. Creating process customer order from elemental business services

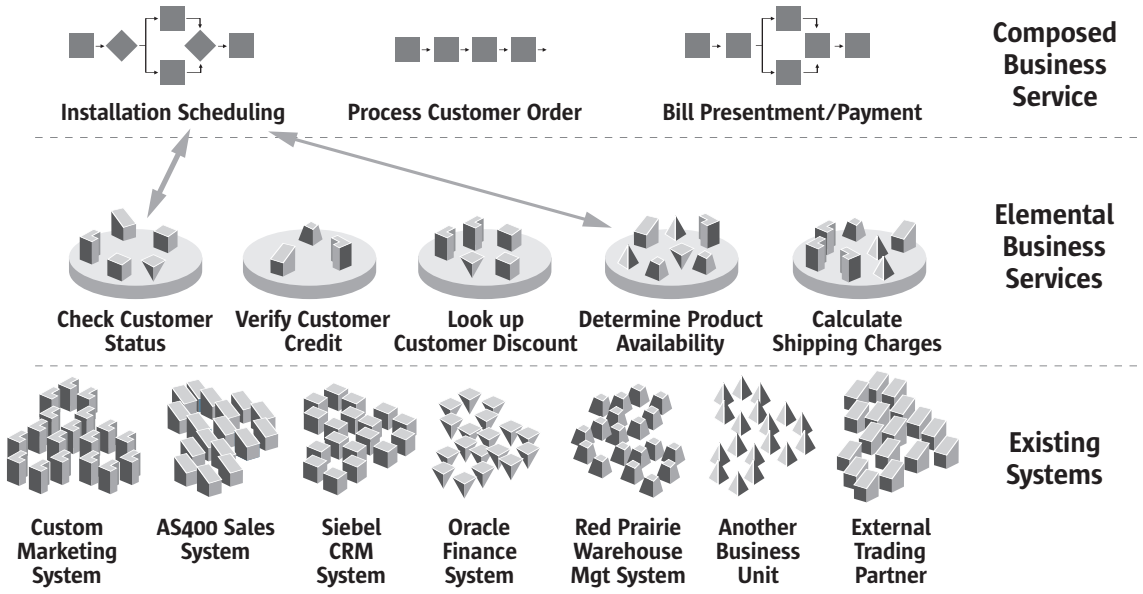

Figure 7. Reusing two elemental business services to create an installation scheduling business service

Further composition can then take place to combine multiple composed business services in order to create composite applications (Figure 8).

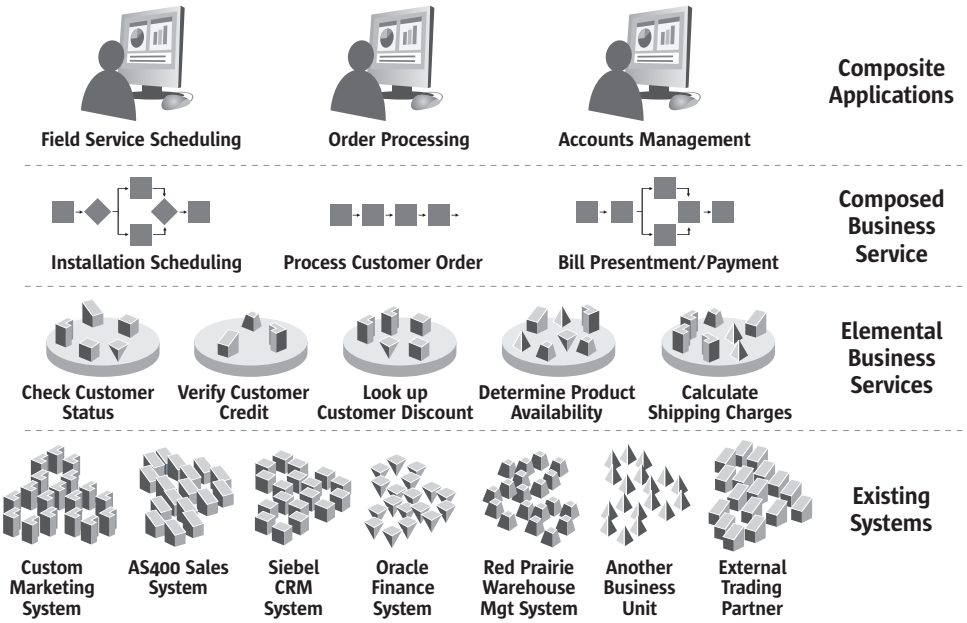

Figure 8. Composite applications are made up of one or several composed business services

The Technology Required to Build an SOA

Given the description of an SOA just presented, it is clear that this structured approach assists application designers in better understanding how to build applications in a modular manner. It is also clear that an SOA presents a readily understandable approach to leveraging application components in order to build multiple related applications with less effort.

So, if an SOA seems to be a good way to approach application design, development, and deployment, how do you go about building the technical infrastructure that is required?

Everything begins with the recognition that composite applications are built on top of existing applications, databases, and files. And, in case you have not noticed, those existing applications, databases, and files were built at different times, by different people, using different technologies, designs, data models, and process semantics.

In other words, perhaps the most foundational assumption in building an SOA is that nothing in the realm of existing application and data assets is exactly the same. If you happen to stumble across two applications that are exactly the same, you have actually found two instances of the same application that were configured and deployed in exactly the same way — a rare occurrence that does nothing to invalidate this approach to building an SOA.

In any event, with all applications built differently, the first thing to do in order to create an SOA is build a layer of wrappers or adapters that can take Customer Information Control System (CICS) and Information Management System (IMS) transactions, Virtual Sequential Access Method (VSAM) and Indexed Sequential Access Method (ISAM) files, relational and nonrelational databases, mainframe and minicomputer programs, as well as legacy and packaged applications, and make them all look the same from a technological perspective (Figure 9). This is the equivalent of traveling with a set of adapters plugs, so that a razor or hair dryer can be plugged into a wall socket in any country.

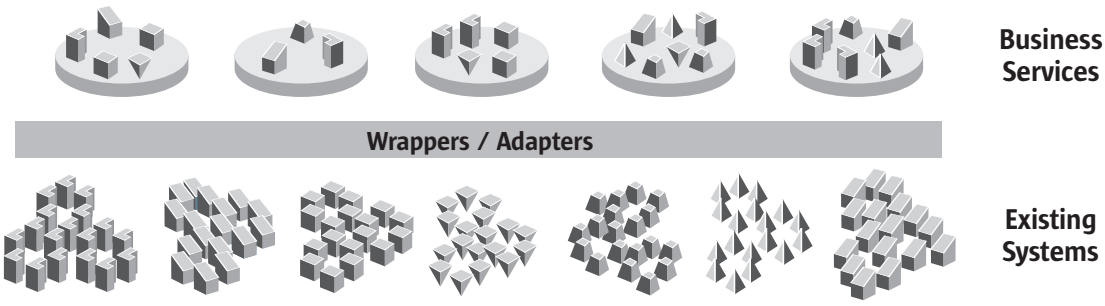

Figure 9. Adapters allow diverse applications and data sources to be accessed in a consistent manner

So adapters take transactions that are exposed as one of a variety of application programming interface (API) technologies and convert them to more standard APIs. For example, these days it is likely that adapters are employed to convert CICS, CORBA, or Business Application Programming Interface (BAPI) calls into Java™ technology, .Net, or Web services calls.

However, this is interoperability, not integration. In other words, while adapters or wrappers move the numbers and letters from source to target applications, they do not ensure that the numbers and letters mean the same thing when they get there (Figure 10).

Sure, the data is sent back and forth and it looks right to both programs. But the data can look right and still mean entirely different things to the programs at either end of the exchange. For example, two programs may both have fields called *item-weight* and, in both cases, the field is defined as a seven-character number with two decimal places — nnnnn.nn. It seems, at first glance, that we are dealing with very similar fields. However, that is not true, because the first program uses item-weight to describe a field that describes the gross weight of the item in kilograms, while the second program uses item-weight to describe the net weight of the items in pounds.

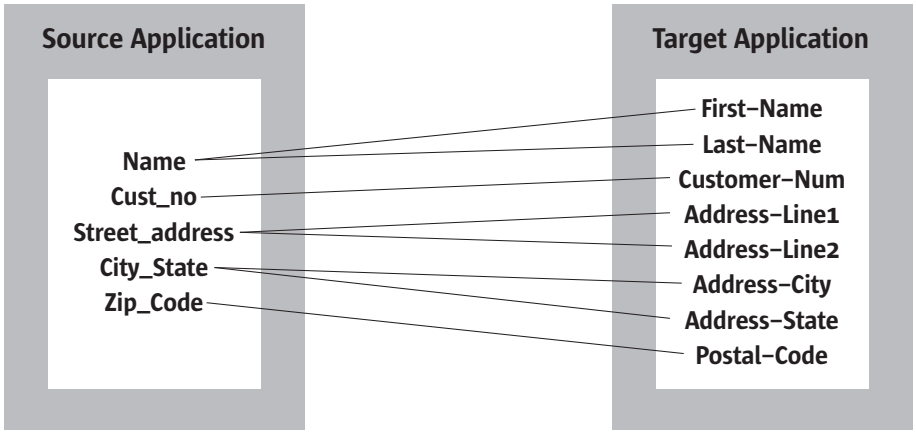

Figure 10. No two applications share the same data format or semantics

Of course, other meaningful differences exist between the representation of similar business objects in the source and target applications. A customer's address in one application may include only five fields, while the customer-address field in a second application might include eight fields.

Or the source and target applications may assemble similar data into different hierarchical structures. Consider a bill of materials for a bicycle. One program may list two wheel rims, two hubs, and 72 spokes, while a second program shows just two wheels (each wheel being made up of one rim, one hub, and 36 spokes).

For these reasons, it is necessary to carefully analyze the data structure in the source and target applications, as well as the context within which each field is used in those programs. Then the data must be transformed accordingly.

That means to provide for true integration, the adapter layer must be paired with a transformation layer (Figure 11).

This transformation layer makes it possible to take different representations of business objects — invoices, letters of credit, bills of materials, and so on — and make them mean exactly the same thing to different programs, regardless of when, where, or for what purpose those programs were written.

In building an SOA, this transformation layer is underappreciated, but it is both essential and difficult. It is not easy to get data transformation right, as the transformation that must take place to build an SOA requires deep knowledge of the format and use of data within all source and target applications.

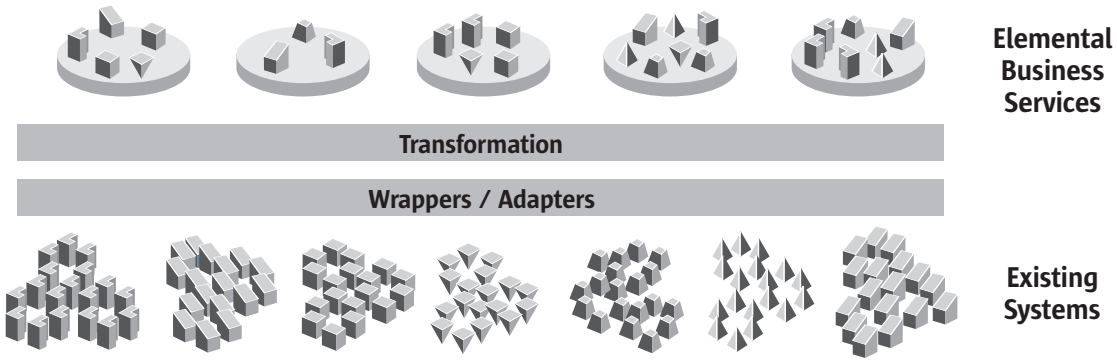

Figure 11. Data transformation converts the format of source data structures, as well as the properties of individual source data elements, so that they can be used by multiple target applications

Once adapters are paired with data transformation, you have the technology that is required to build standard representations of business objects as they are created and maintained within sets of legacy and packaged applications. However, as mentioned earlier, it is necessary to combine these basic business objects to create objects that can readily be used within composite applications (Figure 12). This part of the integration process requires development of microflows in which several processing steps are automated and managed. For instance, you may need to access two internal systems and an external application to aggregate all of the information that is required to expose a customer information business object to customer service, logistics tracking, and order management composite applications.

It is important to note that these microflows are most often stateless, in that a microflow executes the logic required to combine data into composed business logic, and then it wipes its slate clean in preparation for the next task. In other words, a microflow does not store information between instances of activity.

This is different from the stateful business processing characteristic of BPM — the core functionality of many composite applications (Figure 13). This stateful processing is required because the activities that take place at the composite application level are likely to take place over an entirely arbitrary timeframe — it could be hours, days, or weeks before all steps of the composite application's multistep process are completed. This means that it is necessary to take a snapshot of the state of each process instance after the completion of each step in the process flow. Then, that state has to be stored until all the steps in the process instance (which may be an individual order, shipment, or payment) have been completed.

This state management is what separates microflow processing from BPM. While it is possible to write state management logic in a third generation language (3GL) like Visual basic, C++, C#, or Java (it is possible, but not advisable — it takes too much time and requires too much debugging), the state management built into BPM products does all of this for you.

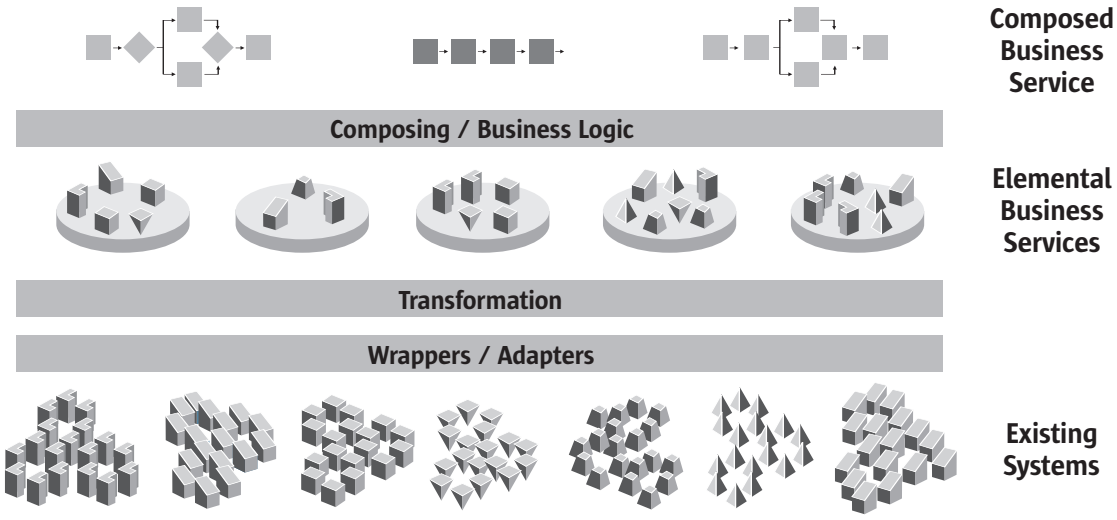

Figure 12. Data and functionality from multiple applications may be combined within microflows to create composed business objects, the fundamental building blocks of composite applications

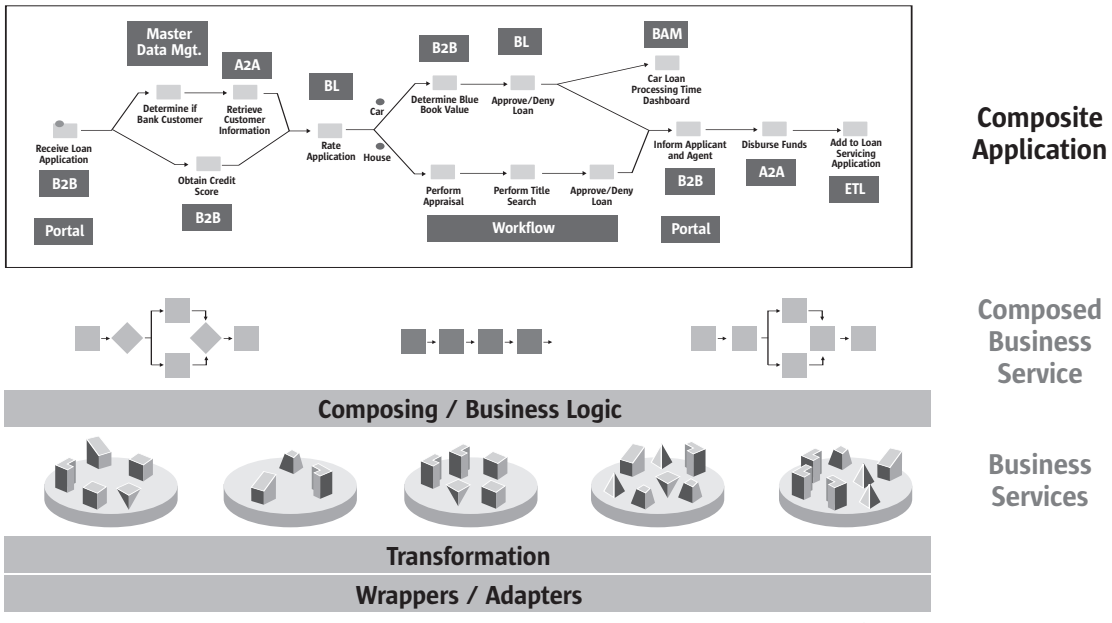

Figure 13. When built on top of a carefully designed SOA, composite applications do not require hand coding. Instead, application integration tools build the links to legacy and package applications while a BPM product is used to define the multistep processing that makes up the rest of the composite application.

Two types of communications are required when building composite applications on top of an SOA. Request/reply middleware supports the real-time interactions between programs, while messaging middleware supports the asynchronous communication that is needed (Figure 14).

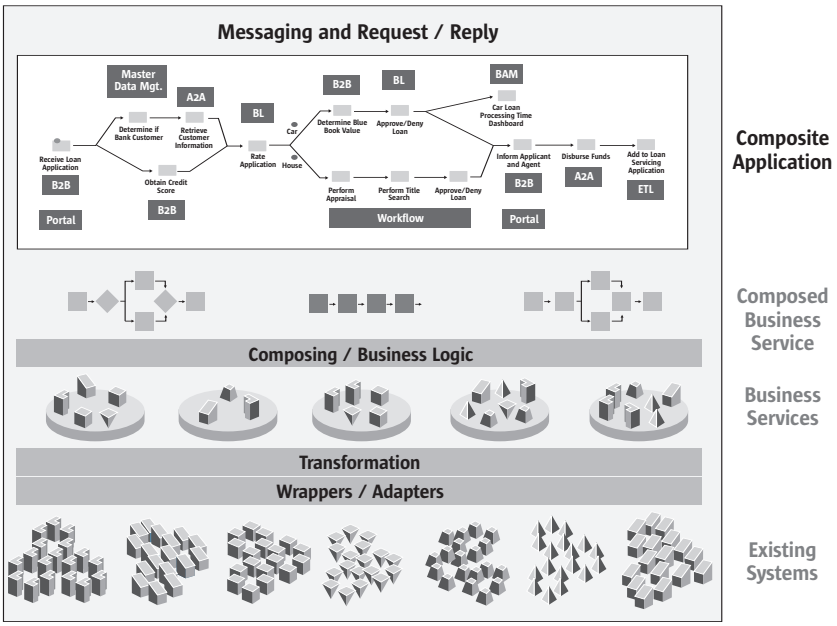

Figure 14. The communication framework in an SOA that enables composite applications must support both request/reply and messaging-based interactions.

The messaging that is required can actually take two different forms. Send/receive addressing is used when the source program sends a message to one or a few known target programs. In contrast, publish/subscribe addressing is used when the source program sends a message to any number of target programs, and the target programs are not known at development time.

At this point, you are probably wondering why we need such an elaborate infrastructure to support the development, deployment, and maintenance of program-to-program links required to create an SOA. Don't Web services remove the need for elaborate application integration tooling, by replacing it with plug-and-play integration? Unfortunately, it is not quite that simple.

For the most part, we are a bit naïve about standards. To many of us, standards are the pot of gold at the end of the rainbow. Portability standards let us write solutions in one environment, then port them to all other environments. Similarly, interoperability standards let us build links between all applications, even though they run on completely different platforms.

So, the early buzz for each new standard promises “Write Once, Run Anywhere” portability or “plug-and-play” interoperability. But, in the real world, vendors have hidden agendas, users are swayed by columnists or industry analysts, and technology evolves much more slowly than we would like or expect.

For these reasons, developers need to know, as early as possible, which standards will achieve universal use, such as Transmission Control Protocol/Internet Protocol (TCP/IP), and which will be abysmal failures, such as the Distributed Computing Environment (DCE). Also, for those standards that will succeed, how long will it take for the standard to achieve critical mass?

The Standards Maturity Model (SMM) described here (Figure 15), can be used to track standards and predict which ones will rule or fail. It can also help developers predict how long it will take a standard to achieve functional completeness and ubiquity, the twin towers of success.

The SMM is similar in some ways to the Software Engineering Institute’s Capability Maturity Model (CMM), which helps IT managers gauge the level of process maturity in their application development efforts. Like the CMM, the SMM has five levels (Figure 15).

Level 5 – “Functionally adequate” version is ubiquitous.

Level 4 – Many applications use “functionally adequate” version.

Level 3 – “Functionally adequate” version of standard is approved.

Level 2 – Version 1.0 of the standard is proposed.

Level 1 – Many recognize the problem.

Level 0 – Few recognize that there’s a problem.

Figure 15. While the slowly maturing Web services standards can provide some assistance in building an SOA, they are neither sufficient nor essential

- At Level 0 of the SMM, few even realize that there is a portability or interoperability problem.
- At Level 1 of the SMM, everyone knows there is a problem, but the solutions (if any) are proprietary.
- At Level 2, vendors or enterprises collaborate on a new standard. The standard is not complete, because there are always gaps and ambiguities in the first attempt. But this early version is submitted to a standards body — such as the World Wide Web Consortium (W3C) or Organization for the Advancement of Structured Information Standards (OASIS) — and comments are invited.
- At Level 3, the standard has gone through several revisions and it is now functionally complete. It addresses the technical issues that its proponents set out to address.
- At Level 4, most new applications and tools implement the standard.
- Full standard maturity is not reached until Level 5. Here, most deployed applications support interoperability standards, and most platforms in use support portability standards.

There are several reasons why most standards never reach Level 5. First, both users and vendors distrust anything that promises to level the playing field. They want to gain an edge on the competition, and standards are often seen as antithetical to that goal. Then, there is the cost of fully embracing a standard. While a standard may be widely used for new development, to reach Level 5, most existing applications and run-time environments must be replaced or retrofitted to accommodate it. That is extremely costly, and the benefit may not be adequate or sufficiently immediate to justify the investment.

So, what does all this say about the current crop of Web services standards? A review of the full suite of Web services-related standards (Figure 16) shows that most of them are far down the stack. The only Web services-related standards that have made it to Level 5 are TCP/IP, Hypertext Transfer Protocol (http), and Secure Sockets Layer (SSL), all of which predated Web services by years. Even the eXtensible Markup Language (XML), which also predates Web services, is only at Level 4 of the SMM.

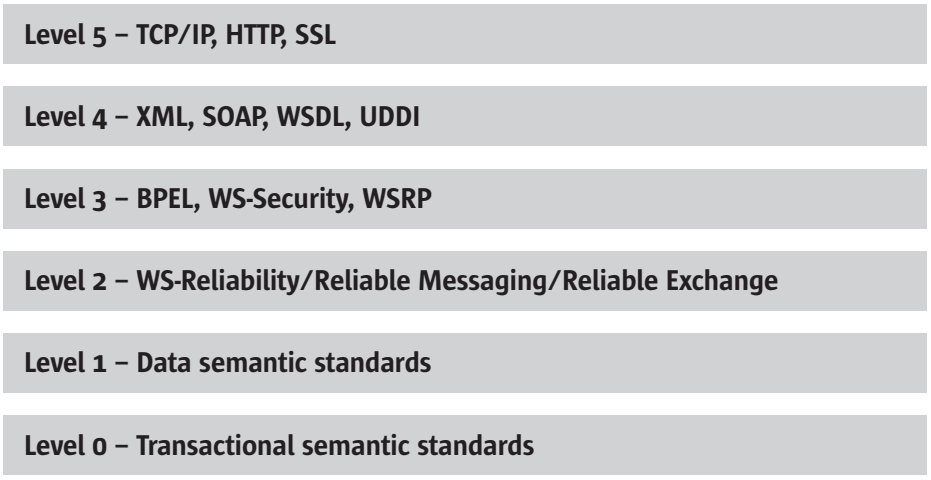

Figure 16. Applying the Standards Maturity Model to Web services standards suggests that it will be many years before Web services provide plug-and-play interoperability for run-the-business applications

The core Web services standards are only now moving from Level 3 to Level 4. They include Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), and Universal Description, Discovery, and Integration (UDDI). They are functionally complete, but they are just at the point where vendors are building support for them into new products.

Unfortunately, for those eagerly anticipating plug-and-play integration enabled by Web services, making it to Level 4 is the easy part. To reach Level 5, thousands of applications must be replaced or reworked. That is a long, expensive, and risky process, unlikely to be completed in a short timeframe. It could be as many as 10 or 15 years before broad, plug-and-play interoperability between applications using SOAP, WSDL, and UDDI becomes a reality.

To make matters worse, these core Web services standards enable only the simplest interoperability scenarios. Standards that govern event recognition, reliable messaging, billing, logging, forwarding, transaction management, and other advanced functions must also be standardized before the more complex interoperability use cases can be built with standard Web services.

But even then, true integration is not enabled. Once the full set of Web services standards become mature and ubiquitous, all that exists is plug-and-play interoperability. In other words, numbers and letters can be exchanged reliably and securely. But source and target applications do not necessarily agree on the semantics of the data that is exchanged. There is no integration until these data semantic issues are addressed.

Also, the source and target applications must agree on the semantics of the transaction. For example, do I respond to your order with a quote and wait for your confirmation? Or, do I just go ahead and send the requested products?

Full plug-and-play integration will occur only when standards address both the relevant interoperability issues *and* semantic issues. Given the diverse ways data is stored and managed within commercial applications, and the many ways organizations govern business transactions enabled by Web services-based information exchanges, Level 5 semantic standards will take even longer than Level 5 interoperability standards.

The bottom line is that plug-and-play integration is far over the horizon. In the meantime, developers must use the best integration technology they can find to implement composite applications on top of an SOA.

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN **Web** sun.com

©2006 Sun Microsystems, Inc. All rights reserved. Sun, Sun Microsystems, the Sun logo, and Java are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd. ORACLE is a registered trademark of Oracle Corporation. Information subject to change without notice. SUNWIN#470927 03/06