JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 8, September-October 2004

From Objects to Services: A Journey in Search of Component Reuse Nirvana

Mahesh H. Dodani, IBM Software, U.S.A.

1 THE SEARCH FOR COMPONENTS

Remember when we started our journey in searching for the "holy grail" of software "integrated circuits"? The Object Oriented (OO) approach provided the promise for developing and using such reusable components to build flexible systems that could be readily adapted to changing requirements.

The continuing journey of using the OO approach as the basis for reusable components for ever increasing complex applications provides insight into the requirements for any supporting architecture, design and technology. OO started with solving some key issues in programming, providing an abstraction that was much more amenable to change than functional components. From programming, we turned our efforts to analysis and design, developing methodologies for identifying objects in the problem domain described by requirements and transitioning these into logical software objects that can be implemented in OO programming languages. From analysis and design methods, we moved on to architectures, design patterns, and frameworks. These allowed us to extend the scope of component reuse to entire enterprise systems.

To reminisce on the OO journey, I went back to look at some of the research and thinking from 10 years ago, and found these interesting nuggets of insight. First were the basic principles that made the OO approach such a good candidate for component reuse:

- Abstraction: A key OO principle was that an object was known by its data type and behavior. This provided the basis for reuse by defining a stable interface for communicating with and using the object.
- Encapsulation: A mechanism for ensuring that the "insides" of an object were protected by the methods and variables that were exposed to the outside. Encapsulation is also a key mechanism for a reusable component as it clearly separates the interface from the implementation.
- Polymorphism: The ability for a single named entity to take on different forms based on the runtime binding to object types. This mechanism facilitated behavior changes to stable interfaces which is key in facilitating reuse.

Cite this column as follows: Mahesh Dodani: "From Objects to Services: A Journey in Search of Component Reuse Nirvana", in *Journal of Object Technology*, vol. 3, no. 8, September-October 2004, pp. 49-54. http://www.jot.fm/issues/issue_2004_09/column5

- Inheritance: The ability for one object to inherit the representation and behavior from another object and to specify additions to the behavior. Originally touted as the main mechanism for reuse, inheritance was later limited to well defined extensions of existing objects.
- Identity: The ability to identify an object and distinguish it from any other object. In the OO world, identity of an object was integral. The key contribution that identity provides to component reuse are the mechanisms to discover components for use regardless of where they were located.

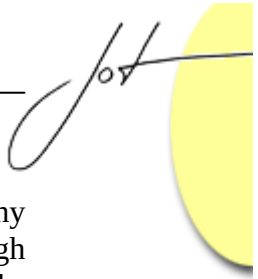
One of the main consequences of component reuse is the impact of changes. To mitigate this impact, the following emerged as driving principles of OO architecture and design:

- Program to an interface, not to an implementation. This principle puts the onus of designing reusable components to the art of designing good interfaces. The journey of OO has provided us support from patterns, frameworks, architectures, and tools to help ensure that good interfaces can be designed and maintained.
- To extend behavior, favor object composition over class inheritance. This principle decouples extended behavior from the original behavior thereby allowing the extended behavior to change without impacting the original behavior.
- Minimize coupling between objects and components to ensure that changes are localized and do not propagate. Again, this priniciple is difficult to quantify and adhere to, and over the years has been supported by patterns, frameworks and architectures.
- Maximize cohesion within an object and within a subsystem to facilitate stable objects and interfaces

As should be clear from the previous paragraphs, a main tenet of reusable components is interface design, and the following define the main principles of good interface design:

- A good interface design:
  - does not expose the underlying attributes (at the class level) or the underlying classes (at the subsystem level),
  - does not expose the underlying implementation, and
  - does one logical unit of work.
- A good system design is one wherein changes can be handled without the need to change interfaces. Following this principle ensures that changes are localized to a class or subsystem in which the change was made, and that such changes do not necessitate changes in clients of the class or subsystem.

Through the last decade, the ever-increasing complexity of systems has tested the limits of the OO architecture and design, and paved the way for newer approaches. The key problem is in handling complex integration of business systems resulting from consolidation of businesses, extending and leveraging mission critical applications across line-of-business silos, building value networks that incorporate partners and other businesses, and participating in dynamic marketplaces. The “paradigm-killer” in this



complex integration problem is that of handling the multiple interfaces of many applications and systems that are distributed over heterogeneous environments. Through our journey, we have gone through many different component architectures, frameworks and middleware that addressed a specific aspect of the problem, from CORBA's attempt to address object interactions over distributed heterogeneous environments to Messaging Oriented Middleware (MOM) to handle communication over multiple interfaces. These complex systems imposed the following requirements on the next evolution of a component reuse approach (see http://www.jot.fm/issues/issue_2003_05/column3 for more in depth coverage):

- Facilitate all aspects of integration, covering at least:
  - People integration, which gives customers, suppliers, partners, and employees a personalized, integrated, anytime/anywhere access to information, transactions, and know-how.
  - Information integration, which federates islands of data to provide a holistic view of a business to users and applications, and that can be efficiently searched or mined for trends.
  - Process integration, which allows a set of applications and services to be choreographed and managed to support a business process.
- Facilitate effective integration of existing applications, systems, and processes incrementally and without the need to "rip and replace".
- Facilitate new and emerging computing needs, e.g. on demand computing (see http://www.computerworld.com/hardwaretopics/hardware/report/0,11188,06282004,00.html.)

As I wrote in an early issue of JOT http://www.jot.fm/issues/issue_2002_07/column5 the service oriented approach was emerging as the main contender in handling this growing complexity.

2 SERVICE ORIENTED ARCHITECTURE

Service Oriented Architecture (SOA) enables flexible integration of applications and resources by:

- representing every application or resource as a service with a standardized interface,
- enabling the services to exchange structured information (messages, documents, 'business objects'), and
- coordinating and mediating between the services to ensure they can be invoked, used and changed effectively.

The key SOA design principles that enables the flexibility of integration is a natural evolution from the OO design principles, and include the following service design principles (see http://wwws.sun.com/software/jini/whitepapers/architecture.html):

- Services are discoverable and dynamically bound.
- Services are self-contained and modular.
- Services stress interoperability.
- Services are loosely coupled.
- Services have a network-addressable interface.
- Services have coarse-grained interfaces.
- Services are location-transparent.
- Services are composable.

So, what is a service? A service represents some functionality (application function, business transaction, system service, etc.) exposed as a component for a business process. The service itself is defined using a well-defined interface that exposes the functionality and hides the underlying implementation details. The service should be stateless to ensure that it is not dependent on the context or state of other services. A service either provides information, or facilitates a change to business data from one valid and consistent state to another. Any dependencies between services should be defined in terms of common business process, function and data models. Services are invoked through defined communication protocols that stress interoperability and location transparency.

Its important here to distinguish between web services and SOA which are two orthogonal yet highly complementary concepts. An SOA design can be built without web services, and web services does not by itself imply conformance to the SOA principles. However, the set of XML-based technologies (shown in Figure 1) that together provide the needed support for enabling web services provide a natural "proof point" of the ability to design and implement an SOA.

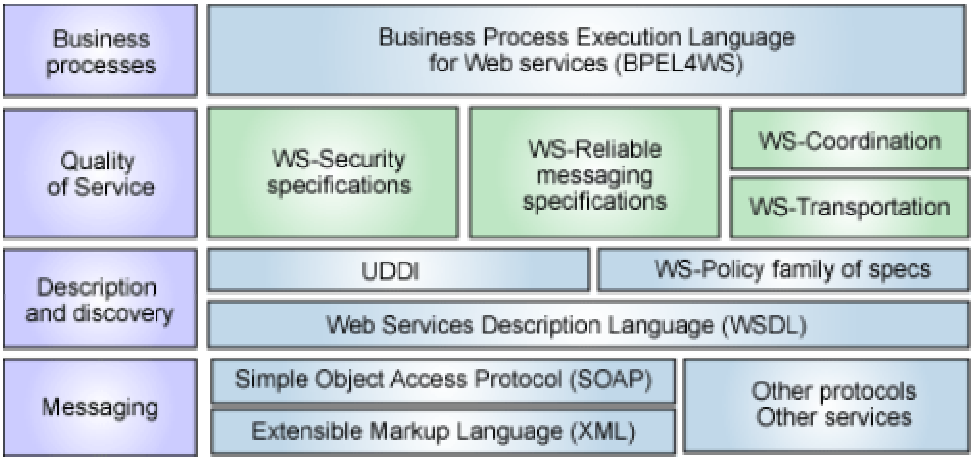
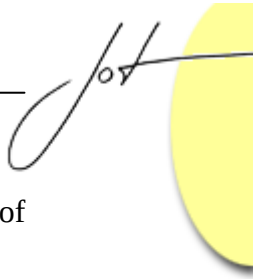


Figure 1: Web Services Technology Stack

Another emerging key component of a SOA design is the Enterprise Service Bus (see http://mediaproducts.gartner.com/reprints/knownow/118991.html.) The Enterprise



Service Bus (ESB) is the service "broker" for SOA and provides the following set of middleware services as shown in Figure 2:

- Communication middleware supporting a variety of protocols, qualities of service, APIs, platforms, and standard protocols.
- A mechanism for injecting intelligent processing of service requests and responses within the network.
- Standard-based tools for enabling rapid integration of services.
- A management system for loosely-coupled applications and their interactions.

While the emerging research on ESB enumerates a wide range of needed capabilities, as pointed out in http://www-106.ibm.com/developerworks/webservices/library/ws-esbscen/ the minimum capabilities required for an ESB to implement a SOA design include the following:

- To facilitate effective service communication, the ESB provides location-transparent routing and addressing services, service addressing and naming administration, support at least one messaging paradigm and transport protocol.
- To facilitate effective service integration, the ESB supports multiple integration mechanisms, including connectors, web services, messaging, and adaptors.
- To facilitate effective service interaction, the ESB provides an open service messaging and interfacing model, that isolates implementations from routing services and transport protocols, and allows implementations to be substituted.

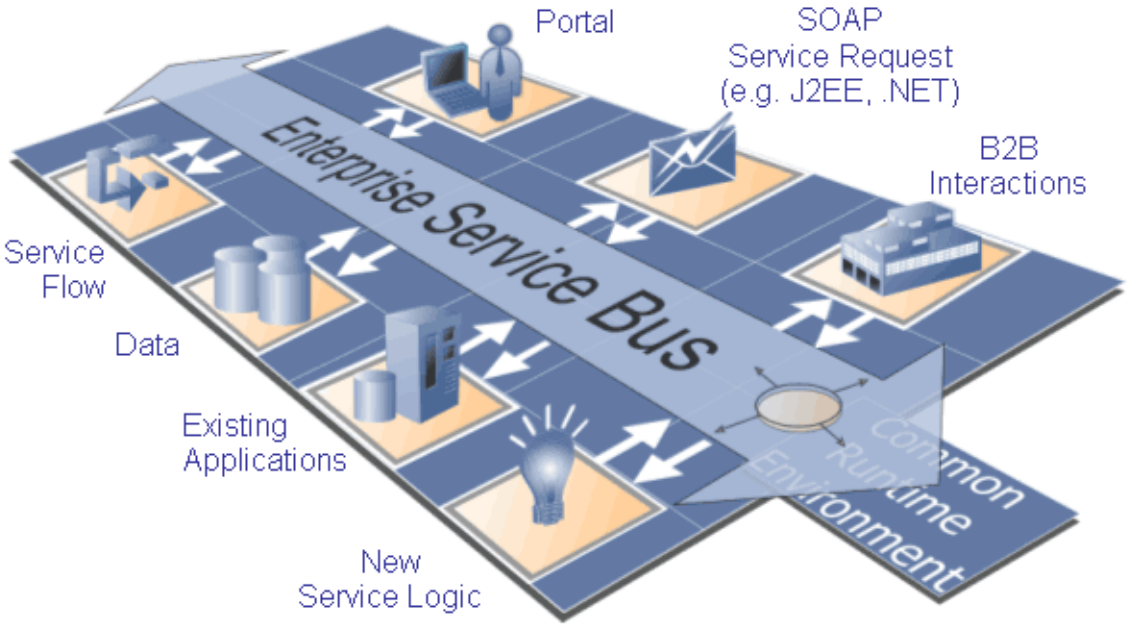


Figure 2: The Enterprise Service Bus

Note that a key ingredient of a SOA and the support provided by the ESB is that all aspects of the enterprise systems need to be implemented as services – this includes services to manage the infrastructure (including servers, storage, security, system

management, etc.) The focus to date has been in addressing the application services to support integration. However, to fully address the emerging system complexity necessitates not only providing flexibility through effective integration but also by simplifying the underlying IT infrastructure. This simplification can also be achieved through the application of the same SOA design to define usable and flexible services that are managed and coordinated through the same model by the ESB.

3 ARE WE THERE YET?

Have we reached the end of our journey in finding component reuse nirvana? Will SOA, web services, and the ESB be capable of handling the current system complexity and any new emerging computing needs and models? As pointed out in http://www.eds.com/thought/thought_leadership_so_architecture.pdf, http://mediaproducts.gartner.com/reprints/knownow/118991.html, and http://roadmap.cbdiforum.com/, the main challenges with the service-oriented approach include some very basic issues, including:

- What is the best granularity for defining a service (coarse or fine grained)?
- Should web services be used for the implementation of all SOA designs?
- Will ESBs support web services or provide support for other protocols?
- How does one assess an implementation to see if it adheres to SOA principles?
- When will products and tools that support web services, ESB and SOA emerge?

Well, as the saying goes “the proof of the pudding is in the eating”, and therefore the test of how well the service-oriented approach will “slay the complexity beast” will be in the experience of applying them and the supporting methodologies, patterns, products and tools that are being developed. Early indications bode well for the service-oriented approach. There is a lot of “pudding” out there, so “happy eating!”

About the author

Mahesh Dodani is a software architect at IBM. His primary interests are in enabling communities of practitioners to design and build complex on demand business solutions. He can be reached at dodani@us.ibm.com.